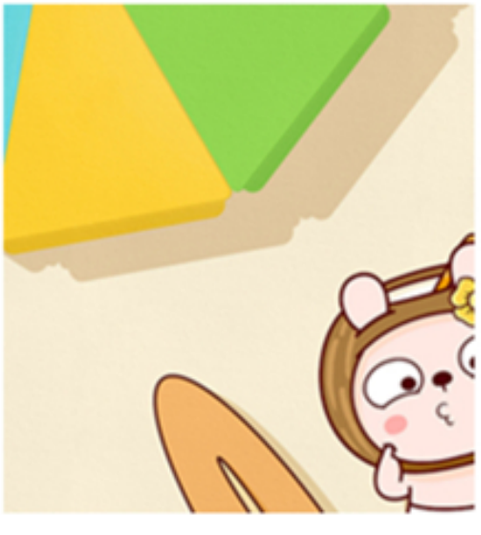
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

EFFECTIVE 系列丛书

华章IT

Pearson

Google高级软件工程师Diomidis Spinellis融合自己30多年系统开发和软件调试的实战经验，深入讨调试的策略、方法、工具和技巧

涵盖调试工作的方方面面，汇聚66条专业调试技巧，分步骤详细讲解，包含大量实用范例代码

Effective Debugging

66 Specific Ways to Debug Software and Systems

Effective Debugging

软件和系统调试的66个有效方法

[希] 迪欧米迪斯·斯宾奈里斯（Diomidis Spinellis）著

爱飞翔 译

机械工业出版社
China Machine Press

非卖品，仅供非商业用途或交流学习使用

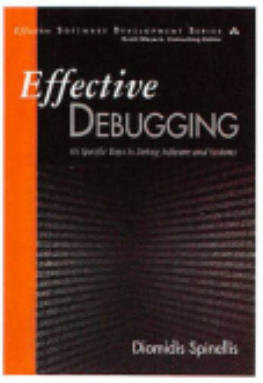
非卖品，仅供非商业用途或交流学习使用

内容简介

Spinellis有超过35年的编程经验，这些经验可以丰富你的调试技术，帮助你选出最适合解决当前问题的办法。他会用多个与具体厂商无关的范例来讲述通用的原则、宏观的策略、具体的技术、高效的工具以及能够提升调试效率的一些做法。

这66条专业的技巧，涵盖调试工作的各个方面，而且全都带有分步的解说与实际的代码，使你能够用来调试各种软件系统，尤其是用来调试那些因分布式组件和服务之间的复杂交互而引发的问题。

无论是单独发生的运行时错误，还是严重的企业系统故障，本书所提供的建议都可以帮你更加迅速且更加顺畅地把调试工作做好。



原书封面

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

华章IT | Information Technology | HZBOOKS

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

EFFECTIVE 系列丛书

Effective Debugging
66 Specific Ways to Debug Software and Systems

Effective Debugging

软件和系统调试的66个有效方法

[希] 迪欧米迪斯·斯宾奈里斯（Diomidis Spinellis）著
爱飞翔 译

机械工业出版社
China Machine Press

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

Effective Debugging：软件和系统调试的66个有效方法 /（希）迪欧米迪斯·斯宾奈里斯（Diomidis Spinellis）著；爱飞翔译．—北京：机械工业出版社，2017.6

（Effective 系列丛书）
书名原文：Effective Debugging: 66 Specific Ways to Debug Software and Systems

ISBN 978-7-111-56889-6

I. E… II. ①迪… ②爱… III. 调试软件 IV. TP311.562

中国版本图书馆 CIP 数据核字（2017）第 090186 号

本书版权登记号：图字：01-2016-8666

Authorized translation from the English language edition, entitled *Effective Debugging:66 Specific Ways to Debug Software and Systems*, 9780134394794, by Diomidis Spinellis, published by Pearson Education, Inc., Copyright © 2017.

All rights reserved.No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2017.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

Effective Debugging
软件和系统调试的 66 个有效方法

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：关 敏 责任校对：殷 虹
印 刷：三河市宏图印务有限公司 版 次：2017 年 6 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：14
书 号：ISBN 978-7-111-56889-6 定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

The Translator's Words 译 者 序

十几年前，我看过 Spinellis 先生所写的《代码阅读》(Code Reading) 和《代码质量》(Code Quality) 两本书，特别喜欢这种全面讲解编程工作中某个领域的教程，这次又读到同一位作者所写的《Effective Debugging》，感觉依然很精彩。

这是一本在思路和技巧上都较为丰富的调试手册。

从思路方面来说，本书介绍了许多宏观与微观的调试办法。例如，在面对软件故障时，既可以从整体情况入手，进行自上而下的调试，也可以从具体故障入手，进行自下而上的调试，还可以考虑用高级的抽象机制、便捷的程序库、直白的算法、简洁的逻辑，乃至另外一门更为合适的编程语言，对 bug 繁多的代码进行改写——这些思路，都能在调试工作中给人以启发。

从技巧方面来说，本书介绍了众多的调试工具与手法：有些可以在程序运行之前，设置断点并对表达式与程序的状态做出断言；有些可以在程序运行之中，将各种调试机

制与程序进行连接，并对其执行情况进行记录；有些则可以在程序崩溃之后，通过核心转储等信息来还原当时的情境。此外，作者还讲解了如何把这些前置、中置和后置技巧与版本控制系统、静态分析工具、动态分析工具及性能测评工具结合起来使用，以提升调试的效率。

全书的8个大类中含有66条技巧，这些技巧都是围绕着“重现bug——探查bug——解决bug”这一主线而展开的。针对这三个阶段，作者进行了详细的分步讲解，给出了很多实用的范例代码与建议，而且特别强调了如何才能稳定地捕获并重现bug，以便给后面两个阶段打下良好的基础。

本书或许还能促使大家思考另外一个问题，那就是：在修复完bug之后，怎样防止有人向程序中引入类似的bug？这可以从代码质量与测试两方面入手。提高代码质量，

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

能够减少程序员对代码的误解，进而降低引入bug的概率；而对测试进行完善，则能够提前捕获很多问题，从而不会使这些问题逐渐积累成复杂的bug。这些理念，作者在书中也时常会提到。

总之，这是一本可以打开思路并拓宽眼界的书籍，大家不妨在自己惯用的调试环境之外，多尝试一下作者所介绍的其他技法、工具和语言，以求达到旧学与新知的融合。

翻译本书的过程中，我得到了机械工业出版社华章公司诸位编辑和工作人员的帮助，在此深表谢意。

由于译者水平有限，不足与疏漏之处请大家发邮件至eastarstormlee@gmail.com，或访问github.com/jeffreybaoshenlee/debugging-errata/issues留言，给我以批评和指教。

爱飞翔

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Preface 前　言

我们在开发软件或对运行软件的系统进行管理的时候，经常会遇到故障。有些故障是因代码问题而引发的编译错误，这种故障可以在短时间内修复；还有一些故障则会使大型系统停机，这将给公司带来每小时数百万的损失（具体货币单位依情况而定）。要想成为一名优秀的专业人士，你就必须在发生故障时迅速找出背后的原因并加以修复。这正是调试的意义所在，也是本书所要谈论的主题。

本书是写给有一定经验的开发者看的，而不是一本介绍性质的读物。它假设读者能够理解用各种编程语言所写成的代码片段，并且会使用高级的 GUI 编程工具以及基于命令行的编程工具。另一方面，我会在书中详细描述调试技巧，因为我发现：即便是对某些开发方法很有经验的编程专家，也依然需要一些手把手的指导，才能够掌握其他的开发方法。此外，如果你已经花了至少几个月时间来调试一些颇具规模的软件，那么应该会更容易理解书中某些高级技巧所适用的场合。

本书所涵盖的范围

本书所要讲解的调试知识，包括与调试有关的策略、工具及方法。我们当前在开发并运作一款复杂的计算系统时，可能会遇到各种问题，而这些调试知识，则使大家能够应对这些问题。过去我们所说的调试，主要是指检测并修复程序错误，而当前却很少

有哪个程序会孤立地运作，即便是一个很小的程序，也会与外部的程序库相链接（通常是动态链接）。更为复杂的程序会运行在应用程序服务器中，会调用 Web 服务，会使用关系型数据库及 NoSQL 数据库，会从目录服务器上获取数据，会运行外部的程序，会利用其他的中间件，也会纳入很多第三方的软件包。于是，要想令整个系统及服务正常地运作，就必须确保其中的组件不会发生故障，这些组件可能是由公司内部人员所开发

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

的，也可能是由第三方所提供的，它们所在的主机或许分布在全球各地。为了应对这种局面，软件开发行业开始重视 DevOps 规程，这套规程旨在同时强调开发者和其他 IT 专业人员所应担负的职责。与之类似，本书想使读者在面对故障时也能够具备这样一种全面的观念，因为在面对一些极为困难的问题时，我们通常无法立刻判断出该问题到底是由哪一个软件组件所引发的。

本书的内容按照从一般到特殊的顺序来进行安排。首先讲解调试策略（第 1 章）、调试方法（第 2 章）以及调试时所用的工具与技术（第 3 章），这些知识使我们能够应对各种软件故障及系统故障。接下来，讨论在调试工作的各个阶段所用到的具体技巧，也就是在使用调试器（第 4 章）、编写程序（第 5 章）、编译软件（第 6 章）以及运行系统（第 7 章）时所用到的调试技巧。与多线程和并发有关的 bug 是很难寻找的，所以最后我们专门用一章（第 8 章）来讲解特定的调试工具及调试技术，使大家能够找出这些 bug。

怎样运用书中的内容

你可以从第一页读起，一页一页往后翻，直到看完。但是别急，其实还有更好的读法。书里给出的建议可以分成以下三种。

- **策略与方法**。这些内容包括我们在面对故障时所应具备的知识以及所应采取的做法。本书第 1 章和第 2 章里面的内容就属于这一类，此外，第 5 章中的很多技巧也可以归入此类。阅读并理解了这些内容之后，你需要在工作中对其加以运用，以便逐渐养成习惯。调试程序的时候，我们需要系统地反思自己所用的办法，如果某个办法行不通，那就应该把自己所经历的路线回顾一遍，这样可以帮助我们发现解决该问题的其他办法。
- **技巧与工具**。这些内容值得大家投入时间去学习，它们主要出现在第 3 章里面，其他章节中的某些内容（如第 36 条）也同样可以归为这一类，我们可以在日常工作中运用这些内容来解决问题。大家应该花时间去学习这些内容，并且要逐步实践它们。这或许意味着我们要放弃自己所熟悉的调试工具，而去使用一些学习曲线较为陡峭但是功能上更加先进的调试工具，那些工具虽然一开始学起

来比较困难，可是从长远来看，却能够帮助你成为调试方面的专家。

- **调试的思路**。当我们遇到困难时，可以根据这些思路来找寻合适的技巧。这些内容不一定每天都会用到，但是当你遇到一个琢磨不透的问题时，它可以帮助

非卖品，仅供非商业用途或交流学习使用

你节省一整天（或者说至少几小时）的时间。比如，如果你不清楚自己所写的C和C++代码为什么无法编译，那么第50条或许能给你一些启发。大家应该快速浏览这些内容，使自己意识到它们可以在某些场合派上用场，等到真正需要使用它们的时候，再去详细研究。

本书对软件开发的其他方面所起的作用

本书里的所有条目都是针对故障的诊断与调试而写的，不过其中有很多建议同样可以用来缩减代码中的bug数量，并且可以使你在遇到这样的bug时能够更为迅速地将其修复。严谨的调试技术与优秀的软件开发方式之间能够形成良性的循环，因此，书中的建议对于你当前或者将来要面对的软件设计、软件构建以及软件管理工作，是可以起到帮助作用的。

设计软件的时候，应该遵循下列建议：

- 使用与其角色相称的高级机制（参见第47条和第66条）。
- 提供调试模式（参见第6条和第40条）。
- 提供对系统操作进行监控与记录的机制（参见第27条、第41条和第56条）。
- 提供一个选项，使得开发者可以用Unix命令行工具来编写与组件有关的脚本（参见第22条）。
- 把内部的错误暴露出来，使其表现为软件故障，而不要将其隐藏起来，使其成为软件中的不稳定因素（参见第55条）。
- 提供一种方式，使得开发者能够在软件发生故障之后获得内存转储（memory dump）信息（参见第35条和第60条）。
- 从数量和范围方面，尽量缩减软件在执行时的不确定因素（参见第63条）。

构建软件的时候，应该遵循下列建议：

- 征求同事的意见（参见第39条）。
- 为你所编写的每个例程创建单元测试（参见第42条）。
- 用断言来验证自己所做的假设是否成立，以及代码的功能是否正确（参见第43条）。
- 尽量把代码写得易于维护，也就是要写出易读、稳定且便于分析和修改的代码（参见第46条和第48条）。

❑ 在构建程序时避免不确定的因素（参见第 52 条）。

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

在对软件的开发及运作进行管理时，应该遵循下列建议（无论是要管理一个团队，还是只管理自己的流程）：

❑ 用适当的事务追踪系统，把遇到的问题记录下来（参见第 1 条）。

❑ 对各种有待处理的事务进行分类，并排定其优先次序（参见第 8 条）。

❑ 把对软件所做的修改适当地记录在修订管理系统中，并且对该系统进行较好的维护（参见第 26 条）。

❑ 渐进地部署软件，使得我们可以在新旧版本之间进行对比（参见第 5 条）。

- 尽量采用各种不同的工具来开发，并试着把程序部署在各种环境中（参见第 7 条）。
- 经常对工具与程序库进行更新（参见第 14 条）。
- 如果使用了第三方的程序库，那么可以考虑购买该程序库的源代码（参见第 15 条）；考虑购买一些较为完善的工具来锁定那些不太容易找到的错误（参见第 51 条、第 59 条、第 62 条、第 64 条及第 65 条）。
- 寻找专门的工具来调试硬件接口及嵌入式系统（参见第 16 条）。
- 使得开发者能够远程调试软件（参见第 18 条）。
- 对于消耗资源较多的故障诊断任务来说，要留出足够的 CPU 及磁盘资源（参见第 19 条）。
- 鼓励开发者之间通过代码评审及编程指导等手段进行协作（参见第 39 条）。
- 鼓励大家进行测试驱动开发（参见第 42 条）。
- 在构建软件的时候，要做性能分析、静态分析以及动态分析（参见第 57 条、第 51 条及第 59 条），并且要打造一套迅速而高效的构建流程与测试流程（参见第 53 条及第 11 条）。

对书中术语的说明

本书所说的 fault（错误）一词，遵循 ISO-24765—2010（Systems and software engineering——Vocabulary）标准，它的意思是："计算机程序里面某个不正确的步骤、流程或数据定义。"这也称为 defect（缺陷）。在日常工作中，我们把这叫做 bug。与之类似，本书所用的 failure（故障）一词，也遵循 ISO-24765—2010 标准，它指的是："因系统或系统组件未能在规定限制之下执行所需功能而引发的事件。"故障可以表现为程序崩溃、程序冻结或是程序的结果有误。故障一词强调的是程序运行的后果，而错误一词强调的则是我们所遇到或使用的某个程序本身有问题。不过有的时候，大家也会用错误及缺陷

这两个词来指代故障，ISO 标准也承认了这一点。笔者在本书中会按照上述定义来区分这几个词，然而在语境比较明确的情况下，我通常也会用问题（problem）一词来指代错误（如"代码有问题"）或故障（如"可以重现的问题"），这样写能够使本书读起来更加流畅，而不至于变成一份法律文件。

Unix 操作系统的 shell、程序库以及工具，目前都可以运行在各种各样的平台上。笔者采用 Unix 一词来指代遵从 Unix 原则及 API 的任何一种系统，包括 Apple 的 Mac OS X、各种 GNU/Linux 发行版（如 Arch Linux、CentOS、Debian、Fedora、openSUSE、Red Hat Enterprise Linux、Slackware 及 Ubuntu）、直接从 Unix 继承而来的系统（如 AIX、HP-UX 及 Solaris）、各种 BSD 衍生系统（如 FreeBSD、OpenBSD 及 NetBSD），以

及运行在 Windows 系统上的 Cygwin。

本书中所列出的 C++、Java 或 Python 代码，针对的也是较新的编程语言版本，不过笔者会避开那些奇怪的或是刚刚推出的特性。

书里面会出现“你的代码”和“你的软件”这两种说法，它们指的是你正在调试的代码以及正在开发的软件。这两种说法听起来比较简洁，而且也暗含了一种对代码所有权的宣示，这对于软件开发人员来说是十分重要的。

笔者所说的例程（routine）一词，是指可供调用的代码单元，如成员函数、方法、函数、过程以及子例程等。

Visual Studio 及 Windows 指的是 Microsoft 公司的相关产品。

修订控制系统（revision control system）及版本控制系统（version control system），是指像 Git 这样能够对软件配置进行管理的工具。

排版约定

Unix 命令行选项使用 --this 这样的形式，与该选项等价的单字母选项使用 -t 这样的形式。Windows 工具中的对应选项，使用 /this 这样的形式。

- 按键采用 Shift-F11 这样的形式。
- 文件路径采用 /etc/motd 这样的形式。
- 菜单操作采用 Debug-New Breakpoint-Break at Function 这样的形式。
- 为了缩短篇幅，笔者会省略 C++ 代码中的 std:: 限定符及 std 名称空间。
- 描述 GUI（图形用户界面）工具时，笔者所说的功能针对的是撰写本书时所能找到的最新版本。如果你所使用的是另外一个版本，那么请参照对应的菜单或窗口来进行操作，也可以在那个版本的文档中查找对应功能的操作办法。值得注意的是，命令行工具的界面几十年来一直都比较稳定，而不像 GUI 工具那样，每个版本都会有一些新的东西。大家可以从这个现象中推出很多结论。

代码及勘误

范例代码及英文原版书的勘误表请参见 www.spinellis.gr/debugging。

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

Acknowledgements 致　谢

首先感谢本书的编辑——Addison-Wesley 出版社的 Trina Fletcher MacDonald，以及 Effective 系列的编辑 Scott Meyers，他们在本书的推进过程中提供了良好的指导与管理。也要感谢本书的技术评审 Dimitris Andreadis、Kevlin Henney、John Pagonis 及 George Thiruvathukal，他们提供了几百条绝妙的思路、评论及建议，使得本书的品质得以改善。特别要感谢文字编辑 Stephanie Geels，她准确地指出书稿中的问题，并适当地加以修改，这个过程本来是令我心生畏惧的，然而她高超的工作水准使该过程变得很轻松。我也要感谢出色完成生产管理工作的 Melissa Panagos、督导整个成书过程的 Julie Nahil，以及为书籍进行排版的 LaTeX 高手 Lori Hughes。Sheri Replin 提出了编辑意见，Olivia Basegio 管理了本书的技术评审组，Chuti Prasertsith 设计了优秀的封面，Stephane Nakib 指导了营销工作，感谢他们，并向 Alfredo Benso、Georgios Gousios 与 Panagiotis Louridas 致意，他们帮助我确立了本书的早期构想。

书中有 4 条是根据我发表在《 IEEE Software 》期刊“ Tools of the Trade”专栏中的材料扩展而成的：

- 第 5 条来自“Differential Debugging” vol.30, no.5, 2013, pp.19-21.
- 第 22 条来自“Working with Unix Tools” vol.22, no.6, 2005, pp.9-11.
- 第 58 条来自“I Spy” vol.24, no.2, 2007, pp.16-17.
- 第 66 条来自“Faking It” vol.28, no.5, 2011, pp.96, 95.

此外还有 4 条需要说明：

- 第 63 条是根据 Martin Fowler 在《 Eradicating Non-Determinism in Tests 》（2014 年 4 月 14 日）及《 TestDouble 》（2006 年 1 月 17 日）这两篇文章中所提出的想法而写成的。

- 第 48 条所提出的大部分重构，都源自 Martin Fowler 所写的《 Refactoring 》（重构）一书（Addison-Wesley, 1999 ）。
- 第 60 条的写作动力，来自《 Real-World Concurrency 》这篇文章（ Bryan Cantrill and Jeff Bonwick, ACM Queue, October 2008 ）。
- 第 66 条中的 Java 代码是根据 Tagir Valeev 的意见写成的。

雅典经贸大学（ Athens University of Economics and Business）的诸位同事，在我的学术生涯中提供了多方面的慷慨帮助，并且（有时是在无形之中）促成了本书的写作，他们包括：Damianos Chatziantoniou、Georgios Doukidis、Konstantine Gatsios、George Giaglis、Emmanouil Giakoumakis、Dimitris Gritzalis、George Lekakos、Panagiotis Louridas、Katerina Paramatari、Nancy Pouloudi、Angeliki Poulymenakou、Georgios Siomkos、Spyros Spyrou 及 Christos Tarantilis。

调试是一种技艺，要在实践中才能学成，因此，我要感谢过去 40 年来所遇到的诸位同事，感谢他们在我所写的代码中发现 bug、给我提供有效的问题报告、对我的代码做出评审及测试，并且教我怎样避免问题、追查问题及修复问题。下面大致按照从近到远的顺序列出我在职场与学术界所遇到的同事：

- 在 Google 的 Ads SRE FE 团队工作时的同事：Mark Bean、Carl Crous、Alexandru-Nicolae Dimitriu、Fede Heinz、Lex Holt、Thomas Hunger、Thomas Koeppe、Jonathan Lange、David Leadbeater、Anthony Lenton、Sven Marnach、Lino Mastrodomenico、Trevor Mattson-Hamilton、Philip Mulcahy、Wolfram Pfeiffer、Martin Stjernholm、Stuart Taylor、Stephen Thorne、Steven Thurgood 及 Nicola Worthington。
- CQS 的同事：Theodoros Evgeniou、Vaggelis Kapartzianis 及 Nick Nassuphis。
- 当前或者曾经在雅典经贸大学管理科学与技术系进行研究和实验的诸位同事：Achilleas Anagnostopoulos、Stefanos Androutsellis-Theotokis、Konstantinos Chorianopoulos、Marios Fragkoulis、Vaggelis Giannikas、Georgios Gousios、Stavros Grigorakakis、Vassilios Karakoidas、Maria Kechagia、Christos Lazaris、Dimitris Mitropoulos、Christos Oikonomou、Tushar Sharma、Sofoklis Stouraitis、Konstantinos Stroggylos、Vaso Tangalaki、Stavros Trihias、Vasileios Vlachos 及 Giorgos Zouganelis。
- 担任希腊财政部的信息系统秘书长时的诸位同事：Costas Balatos、Leonidas Bogiatzis、Paraskevi Chatzimitakou、Christos Coborozos、Yannis Dimas、

Dimitris Dimitriadis、Areti Drakaki、Nikolaos Drosos、Krystallia Drystella、Maria Eleftheriadou、Stamatis Ezovalis、Katerina Frantzeskaki、Voula Hamilou、Anna Hondroudaki、Yannis Ioannidis、Christos K.K.Loverdos、Ifigeneia Kalampokidou、Nikos Kalatzis、Lazaros Kaplanoglou、Aggelos Karvounis、Sofia Katri、Xristos Kazis、Dionysis Kefallinos、Isaac Kokkinidis、Georgios Kotsakis、Giorgos Koundourakis、Panagiotis Kranidiotis、Yannis Kyriakopoulos、Odyseas Kyriakopoylos、Georgios Laskaridis、Panagiotis Lazaridis、Nana Leisou、Ioanna Livadioti、Aggeliki Lykoudi、Asimina Manta、Maria Maravelaki、Chara Mavridou、Sofia Mavropoulou、Michail Michalopoulos、Pantelis Nasikas、Thodoros Pagtzis、Angeliki Panayiotaki、Christos Papadoulis、Vasilis Papafotinos、Ioannis Perakis、Kanto Petri、Andreas Pipis、Nicos Psarrakis、Marianthi Psoma、Odyseas Pyrovolakis、Tasos Sagris、Apostolos Schizas、Sophie Sehperides、Marinos Sigalas、George Stamoulis、Antonis Strikis、Andreas Svolos、Charis Theocharis、Adrianos Trigas、Dimitris Tsakiris、Niki Tsouma、Maria Tzafalia、Vasiliki Tzovla、Dimitris Vafiadis、Achilleas Vemos、Ioannis Vlachos、Giannis Zervas 及 Thanasis Zervopoulos。

- 在 FreeBSD 项目工作时的同事：John Baldwin、Wilko Bulte、Martin Cracauser、Pawel Jakub Dawidek、Ceri Davies、Brooks Davis、Ruslan Ermilov、Bruce Evans、Brian Fundakowski Feldman、Pedro Giffuni、John-Mark Gurney、Carl Johan Gustavsson、Konrad Jankowski、Poul-Henning Kamp、Kris Kennaway、Giorgos Keramidas、Boris Kovalenko、Max Laier、Nate Lawson、Sam Leffler、Alexander Leidinger、Xin Li、Scott Long、M.Warner Losh、Bruce A.Mah、David Malone、Mark Murray、Simon L.Nielsen、David O'Brien、Johann'Myrkraverk'Oskarsson、Colin Percival、Alfred Perlstein、Wes Peters、Tom Rhodes、Luigi Rizzo、Larry Rosenman、Jens Schweikhardt、Ken Smith、Dag-Erling Smørgrav、Murray Stokely、Marius Strobl、Ivan Voras、Robert Watson、Peter Wemm 及 Garrett Wollman。
- LH Software 及 SENA 的同事：Katerina Aravantinou、Michalis Belivanakis、Polina Biraki、Dimitris Charamidopoulos、Lili Charamidopoulou、Angelos Charitsis、Giorgos Chatzimichalis、Nikos Christopoulos、Christina Dara、Dejan Dimitrijevic、

Fania Dorkofyki、Nikos Doukas、Lefteris Georgalas、Sotiris Gerodianos、Vasilis Giannakos、Christos Gkologiannis、Anthi Kalyvioti、Ersi Karanasou、Antonis Konomos、Isidoros Kouvelas、George Kyriazis、Marina Liapati、Spyros Livieratos、Sofia Livieratou、Panagiotis Louridas、Mairi Mandali、Andreas Massouras、Michalis Mastorantonakis、Natalia Miliou、Spyros Molfetas、Katerina Moutogianni、Dimitris Nellas、Giannis Ntontos、Christos Oikonomou、Nikos Panousis、Vasilis Paparizos、Tasos Papas、Alexandros Pappas、Kantia Printezi、Marios Salteris、Argyro Stamati、Takis Theofanopoulos、Dimitris Tolis、Froso Topali、Takis Tragakis、Savvas Triantafyllou、Periklis Tsahageas、Nikos Tsagkaris、Apostolis Tsigkros、Giorgos Tzamalis 及 Giannis Vlachogiannis。

- 欧洲计算机产业研究中心（European Computer Industry Research Center，ECRC）的同事：Mireille Ducassé、Anna-Maria Emde、Alexander Herold、Paul Martin 及 Dave Morton。
- 伦敦帝国学院（Imperial College London）计算机科学系的同事：Vasilis Capoyleas、Mark Dawson、Sophia Drossopoulou、Kostis Dryllerakis、Dave Edmondson、Susan Eisenbach、Filippos Frangulis Anastasios Hadjicocolis、Paul Kelly、Stephen J.Lacey、Phil Male、Lee M.J.McLoughlin、Stuart McRobert、Mixalis Melachrinidis、Jan-Simon Pendry、Mark Taylor、Periklis Tsahageas 及 Duncan White。
- 加州大学伯克利分校计算机科学研究组（Computer Science Research Group，CSRG）的同事：Keith Bostic。
- Pouliadis&Associates 公司的同事：Alexis Anastasiou、Constantine Dokolas、Noel Koutlis、Dimitrios Krassopoulos、George Kyriazis、Giannis Marakis 及 Athanasios Pouliadis。
- 在各种会议及各种场合认识的诸位朋友：Yiorgos Adamopoulos、Dimitris Andreadis、Yannis Corovesis、Alexander Couloumbis、John Ioannidis、Dimitrios Kalogeras、Panagiotis Kanavos、Theodoros Karounos、Faidon Liampotis、Elias Papavassilopoulos、Vassilis Prevelakis、Stelios Sartzetakis、Achilles Voliotis 及 Alexios Zavras。

最后感谢家人多年来对我的支持，以及在我撰写本书时所给予的鼓励，有时我要在家里调试系统，甚至放弃度假和周末休息时间而去写稿。尤其感谢 Dionysis 帮我绘制图 5.2，并感谢 Eliza 和 Eleana 帮我选定本书的封面。

Contents 目　录

译者序
前言
致谢

第 1 章　宏观策略 ····· 1

第 1 条：通过事务追踪系统处理所有的问题 ····· 1
第 2 条：在网上确切地查询你所遇到的问题，以寻求解决问题的灵感 ····· 4
第 3 条：确保前置条件与后置条件都能够得到满足 ····· 6
第 4 条：从具体问题入手向上追查 bug，或从高层程序入手向下追查 bug ····· 7
第 5 条：在能够正常运作的系统与发生故障的系统之间寻找差别 ····· 9
第 6 条：使用软件自身的调试机制 ····· 13
第 7 条：试着用多种工具构建软件，并将其放在不同的环境下执行 ····· 16
第 8 条：把工作焦点放在最为重要的问题上 ····· 20

第 2 章　通用的方法与做法 ····· 23

第 9 条：相信自己能够把问题调试好 ····· 23
第 10 条：高效地重现程序中的问题 ····· 26
第 11 条：修改完代码之后，要能够尽快看到结果 ····· 29
第 12 条：将复杂的测试场景自动化 ····· 30
第 13 条：使自己尽可能多地观察到与调试有关的数据 ····· 32
第 14 条：考虑对软件进行更新 ····· 34

第 15 条：查看第三方组件的源代码，以了解其用法 …… 35
第 16 条：使用专门的监测及测试设备 …… 37
第 17 条：使故障更加突出 …… 40
第 18 条：从自己的桌面计算机上调试那些不太好用的系统 …… 42
第 19 条：使调试任务自动化 …… 44
第 20 条：开始调试之前与调试完毕之后都要把程序清理干净 …… 46
第 21 条：把属于同一个类型的所有问题全都修复好 …… 47

第 3 章　通用的工具与技术 …… 49

第 22 条：用 Unix 命令行工具对调试数据进行分析 …… 49
第 23 条：掌握命令行工具的各种选项及习惯用法 …… 55
第 24 条：用编辑器对调试程序时所需的数据进行浏览 …… 57
第 25 条：优化工作环境 …… 59
第 26 条：用版本控制系统寻找 bug 发生的原因及经过 …… 64
第 27 条：用工具监测由多个独立程序所构成的系统 …… 67

第 4 章　调试器的使用技巧 …… 71

第 28 条：编译代码时把符号信息包含进来，以便于调试 …… 72
第 29 条：对代码进行单步调试 …… 76
第 30 条：设置代码断点和数据断点 …… 77
第 31 条：了解反向调试功能 …… 80
第 32 条：查看例程之间的相互调用情况 …… 83
第 33 条：查看变量及表达式的值，以寻找程序中的错误 …… 84
第 34 条：了解怎样把调试器连接到正在运行的进程上 …… 87
第 35 条：了解怎样运用核心转储信息来进行调试 …… 89
第 36 条：把调试工具设置好 …… 92
第 37 条：学会查看汇编代码及原始内存 …… 95

第 5 章　编程技术 …… 100

第 38 条：对可疑的代码进行评审，并手工演练这些代码 …… 100

非卖品，仅供非商业用途或交流学习使用

第 39 条：审读代码并与同事讨论 …… 102
第 40 条：给软件添加调试机制 …… 103
第 41 条：添加日志语句 …… 107
第 42 条：对软件进行单元测试 …… 111
第 43 条：用断言进行调试 …… 114
第 44 条：改动受测程序，以验证自己的推想 …… 118
第 45 条：尽量缩小正确范例与错误代码之间的差距 …… 119
第 46 条：简化可疑代码 …… 120
第 47 条：将可疑代码改用另外一种编程语言来写 …… 123
第 48 条：改善可疑代码的可读性与结构 …… 124
第 49 条：要清除 bug 的根源，而不仅仅消除其症状 …… 128

第 6 章　编译时的调试技术 …… 130

第 50 条：对生成的代码进行检视 …… 130
第 51 条：使用静态程序分析工具 …… 133
第 52 条：对项目进行配置，令程序能够以固定的方式构建和执行 …… 138
第 53 条：对调试所用程序库及构建代码时所应执行的检查进行配置 …… 141

第 7 章　运行时的调试技术 …… 147

第 54 条：通过构建测试用例来寻找错误 …… 147
第 55 条：令软件在遇到问题时尽早退出 …… 151
第 56 条：检视应用程序的日志文件 …… 152
第 57 条：对系统和进程所执行的操作进行性能评测 …… 156
第 58 条：追踪程序的执行情况 …… 160
第 59 条：使用动态程序分析工具 …… 166

第 8 章　调试多线程的代码 …… 169

第 60 条：通过事后调试来分析死锁问题 …… 169
第 61 条：捕获并重现 …… 176
第 62 条：用专门的工具来探查死锁与竞争条件问题 …… 180

非卖品，仅供非商业用途或交流学习使用

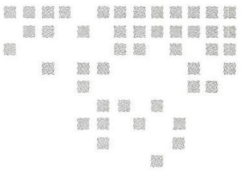
非卖品，仅供非商业用途或交流学习使用

第 63 条：把不确定的因素隔离出来，或将其移除 …… 186
第 64 条：检查资源争用情况，以解决与可伸缩性有关的问题 …… 188
第 65 条：用性能计数器寻找伪共享问题 …… 191
第 66 条：考虑用更为高级的抽象机制来重写代码 …… 195

网上资源 …… 203

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



第 1 章 Chapter 1

宏观策略

要想解决问题，就必须先选定最佳的策略，这样才可以事半功倍。如果当前的策略无法奏效，那就应该立刻改用成功率第二高的办法。

第 1 条：通过事务追踪系统处理所有的问题

请想象这样一种场景：George 在电话里朝你大吼，说你开发的那个应用程序“运行不了”，于是你赶紧把问题写在便签上，然后把它贴在显示器旁边，显示器周围还有很多类似的纸条。现在你开始回想，自己到底有没有把新版程序所需的最新库文件发给 George。实际上，我们不应该这样来处理问题，而是应该改用下面的办法。

首先，要保证有一套事务追踪系统（issue-tracking system）可供使用。很多开源软件库，如 GitHub 和 GitLab 等，都提供基本的事务追踪系统，该系统与它们所提供的其他功能是集成在一起的。有些组织使用一种名为 JIRA 的专有系统，这种系统要复杂得多，它可以在企业内部运行，也可以作为服务来运行。还有一些组织使用开源替代品，如 Bugzilla、Launchpad、OTRS、Redmine 或者 Trac。选择哪个系统并不重要，重要的是必须保证所有的事务都记录在这个系统里面。

如果某个问题没有记录在事务追踪系统中，那我们就拒绝处理该问题。坚持使用这

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

样的系统，能够带来下面几个好处：

- 可以看见调试工作所取得的进展。
- 可以对软件的发行进行追踪与规划。
- 帮助我们确定各种工作项（work item）之间的优先次序。
- 帮助我们把常见的事务及其解决方案整理成文档。
- 防止我们遗漏某些问题。
- 可以自动生成发行说明（release note）。
- 可以用作知识库，使我们对软件中的缺陷进行估量及反思，并从中总结经验。

对于公司里面那些不必亲自汇报问题的高层员工来说，你可以代他们汇报问题。如果某个问题是你自己发现的，那你也可以自己把这个问题提交到系统里面。有一些公司规定：在修改代码之前，必须先指明这次修改所涉及的事务。

我们还要保证的是：每一项事务都能够精确地描述问题的重现方式。最好能在其中给出一个简短（short）、自足（self-contained）且正确（correct，也就是可以正确编译并运行）的例子（example），即 SSCCE。我们可以把这个例子直接剪下来，粘贴到应用程序中，以便重现它所要说明的问题（参见第 10 条）。为了使大家能够写出有效的错误报告（bug report），我们应该制订一份规范，并劝说所有人都认真遵照这份规范来撰写报告。（我看到有一家公司把这些规范贴在厕所门上。）

此外，错误报告还必须具备精确的标题（precise title），并写明 bug 的优先级（priority）、严重程度（severity）、受影响的利益相关者（stakeholder），以及该 bug 的发生情境（environment）。在填写这些内容时，要注意以下几点：

- 精准的标题使我们能够在事务汇总报告中迅速找出这个 bug。用“程序崩溃”这几个字做标题是很糟糕的，应该改成“正在保存时单击刷新按钮，会使程序崩溃”。
- 严重程度能够帮助我们判断 bug 的优先级。与数据丢失有关的问题当然是很严重的，而另外一些无关紧要的问题，或是可以用某种明确的方式来绕过的问题，则显得不那么严重。团队可以根据 bug 的严重程度来对清单里面的各项事务进行分类，以决定哪些事务需要立刻解决，哪些可以稍后解决，哪些应该忽略。
- 对事务进行分类并排定其次序之后，就可以把结果填写在优先级这一栏中了，这使得我们能够据此安排各项事务的处理顺序（参见第 8 条）。在很多项目中，

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

bug 的优先级是由开发者或项目主管来设置的，如果允许终端用户来设置，那么他们总是会把自己提交的所有 bug 都设置成最高优先级。虽说管理人员、客户代表、其他团队的开发者以及销售人员都宣称自己提交的事务应该最先得到处理，但我们还是应该根据实际情况来设置优先级。

- 在事务中指出受到影响的利益相关者，可以帮助团队获知与该事务有关的其他一些信息，并帮助产品拥有者来决定事务的优先次序。有些公司甚至会在利益相关者后面标注他们给公司带来的年度收入。（例如，“由 Acme 所提交，该客户给公司带来的年度收入是 25 万元。”）
- 对于某些难以捕获的 bug 来说，情境描述可以提供线索，使得我们能够重现这种 bug。不要强迫用户填写过多的信息，例如，PC 的序列号、BIOS 的日期，以及系统中每一个程序库的版本等，这样做会令用户觉得非常麻烦，从而跳过这些内容。我们只应该询问与 bug 密切相关的那些细节，对于 Web 应用程序来说，浏览器的信息自然是相当重要的，而对于移动应用程序来说，我们或许想知道设备的制造商及型号。如果能够通过软件来自动提交这些信息，那就更好了。

使用事务追踪系统时，我们一定要通过它来记录进度。大部分追踪系统都允许用户在每个事务后面持续追加各种形式的评论。这些文档可以把调查及修复 bug 时所经历的步骤记录下来，其中也可以包括修复 bug 时所遇到的困境。这样做可以使公司内的工作更加透明。我们应该精确地写出记录或追踪程序行为时所执行的各种命令，这样做很有用，因为明天你可能就要重新执行这些命令，你或你的同事也有可能要在一年之后寻找一个类似的 bug。当你辛苦地完成了为期一周的 bug 搜寻工作之后，这些笔记可以帮助你回顾工作内容，使你能够更好地把这些天所做的事情解释给团队或管理者听。

要点

- 通过事务追踪系统来处理所有的问题。
- 确保每项事务都能够以短小、自足而又正确的范例（SSCCE），精确地描述出该问题的重现方式。
- 对事务进行分类，并根据每项事务的优先级与严重程度来安排工作。
- 通过事务追踪系统来记录进度。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

第 2 条：在网上确切地查询你所遇到的问题，以寻求解决问题的灵感

现在很少有哪个工作场所不能上网，如果在一个无法上网的地方开发程序，那我的效率会很低。遇到代码错误的时候，我们应该上网搜索，或者与同事一起寻找解决办法。

有一个相当有效的搜索技巧，是把由第三方组件所给出的错误消息打上双引号，并将其粘贴到浏览器的搜索框里面。把待搜索的内容放在一对双引号中，意思是要告诉搜索引擎：只搜索与该内容精确匹配的页面。这样做可以使搜索结果更加准确。还有一个很有用的技巧，是把与错误有关的程序库或中间件的名称、对应的类名或方法名，以及所返回的错误代码，也一并放在搜索框里面。要查找的函数名称越罕见，搜索到的结果也就越确切，例如，搜寻 PlgBlt 所得到的结果，要比搜寻 BitBlt 好得多。此外，我们也应该试着搜索意思相近的词，例如，除了搜索“hangs”（挂起），还可以搜索“freezes”（冻结），除了搜索“disabled”（禁用），还可以搜索“grayed”（变灰）。

要想解决一些与 API 调用有关的难题，我们通常可以观察其他人是如何使用这些 API 的。我们可以看看开源软件如何使用某个函数，如何对传给该函数的参数进行初始化，以及如何解读函数所传回的结果。在这种情况下，专门用来搜索代码的引擎（如 Black Duck Open Hub Code Search），要比 Google 那样的通用引擎更好。例如，如果在这个搜索引擎[⊖]里面查找 mktime，并且只看与某个项目有关的代码，而过滤掉程序库的声明及定义，那我们就会发现下面这样的代码片段：

```
nowtime = mktime(time->tm_year+1900, time->tm_mon+1,
     time->tm_mday, time->tm_hour, time->tm_min,
     time->tm_sec);
```

通过上述代码片段，我们可以看出：mktime 函数与 localtime 函数有所不同，它要求传入的年份必须是完整的数值，而不是距离 1900 年的偏移量，而且它的月份是从 1 开始计算的。这两个地方经常会出错，对于那些没有仔细阅读函数文档的人来说，更是容易在调用时传入错误的参数。

在查看由搜索引擎所给出的结果时，我们要注意这些结果是从哪个网站抓取到的。StackExchange 旗下的网站（如 Stack Overflow）通过很多措施来鼓励用户进行有效的交

⊖ 该引擎现已停止服务，可以考虑改用 searchcode.com 等引擎。——译者注

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

流，因此，在由搜索引擎所给出的结果中，有很多比较切题的讨论及答案都来自这个系列的网站。在浏览 Stack Overflow 上面的答案时，不仅要看提问者所接受的那个回答，而且还要看看其他那些赞同数量比较高的回答。除了答案的正文，我们还可以关注答案下面的评论，因为很多人都会通过评论的方式来给出新的消息，例如，有人会在评论中告诉大家，自己发现了一个可以避免错误的新办法。

如果你把自己精心构造的关键词放入搜索引擎之后，并没有得到有用的结果，那么或许意味着你找错了目标。对于常见的程序库与软件来说，你不太可能成为第一个遭遇某问题的人，因此，如果在网上找不到类似的描述，那可能说明你对问题的判断发生了偏差。例如，你本来以为程序崩溃的原因是某个 API 函数的实现有 bug，但实际上却是传入的日期有误。

如果网上找不到答案，那你可以在 Stack Overflow 网站提问，把自己所面对的问题描述出来，然而，这需要花一定的时间来构建一个简单、自足且正确的范例（SSCCE）。凡是在论坛发问，都应该遵循该 SSCCE 原则，也就是要给出一个其他人可以直接复制、粘贴并编译的例子，使得他们能够看到你所经历的问题（参见第 10 条）。对于某些编程语言来说，甚至可以把范例代码嵌入 SourceLair 或 JSFiddle 这样的在线 IDE，令大家能够在网上直接看到运行效果。sscce.org 网站详细解释了应该怎样针对具体的语言和技术来构造良好的范例。Eric Raymond 所写的文章《How To Ask Questions The Smart Way》也与这个话题有关，值得一读。

笔者发现：只要我能够恰当地描述问题，并且附上合适的范例，那么该问题的解决方案通常就会自然地浮现出来。就算我自己找不到答案，这样的问题也可以吸引一些懂行的人过来进行试验，他们或许能找到办法。

如果你所遇到的问题在某种程度上与开源的软件库或程序有关，而且你认为它们的代码中很可能有 bug，那么可以联系其开发者。常见的做法应该是访问那个开源项目的 bug 追踪系统，并在上面提交一项事务。提交的时候，也应该首先确保其他人没有报告过类似的 bug，并且要把重现该问题的详细步骤准确地写进去。如果那款软件没有 bug 追踪系统，那你可以给它的作者发邮件，邮件要写得相当谨慎，措辞要得体、语气要谦和，因为大部分开源软件的开发者都不是你的雇工。

要点

- 把错误消息打上双引号，以便在网上准确地进行搜索。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 认真查看 StackExchange 系列网站上面的回答。
- 如果上述两种办法都不见效，那你可以自己提问或提交事务。

第 3 条：确保前置条件与后置条件都能够得到满足

修理电子设备的时候，我们首先要检查供电是否正常，也就是检查电流有没有从电源模块正确地流入该设备的电路中。在很多情况下，这项检查都能帮助我们找出问题。计算机程序与之类似，很多问题也可以通过对例程的入口点（entry point）与出口（exit）进行检查而得以确定。入口点就是前置条件（precondition），它指的是程序在即将执行例程时所具备的状态，以及传递给该例程的输入值，出口则是后置条件（postcondition），它指的是程序执行完例程之后的状态及其返回值。如果前置条件得不到满足，那说明用来设置这些前置条件的代码里面有错误，若是后置条件得不到满足，则说明该例程本身有问题。如果两者都正确，那么应该转向其他地方去寻找 bug。

我们可以在例程开始的地方、调用例程的地方或关键算法开始执行的地方设置断点（参见第 30 条）。为了判断前置条件是否得到满足，我们应该仔细检查算法的参数，包括传入的参数值，调用方法时所针对的对象，以及可疑代码所使用的全局状态。尤其要注意以下几点：

- 找出那些本来不应为 null，但实际上却为 null 的值。
- 调用数学函数的时候，确保传入的值位于该函数的定义域之内，例如，调用 log 函数时传入的值应该大于 0。
- 查看对象、结构体与数组的内部细节，确保其内容符合要求。这也可以帮你查出无效的指针。
- 检查变量的取值是否在合理范围之内。如果变量具有 6.89851e-308 或 61007410 这样的可疑取值，那通常表明它还没有初始化。
- 检查传给例程的数据结构是否正确，例如，map 有没有包含预期的键与值，双向链表（doubly linked list）能不能正确地遍历。

然后，我们应该在例程结束的地方、调用完例程的地方或关键算法执行完毕的地方设置断点，以判断该例程的执行效果是否正确：

- 计算出来的结果看上去合理吗？有没有处在预期的范围之内？
- 如果结果合理，而且位于预期的范围之内，那么实际的值是否正确？我们可以

通过手算来演练相应的代码，以验证计算机的执行结果是否正确（参见第38条），也可以将执行结果与已知的正确值相对比，或是采用其他工具或方法来进行验算。

- 例程的副作用是否符合预期？可疑代码所接触到的其他数据是否遭到破坏或拥有了不正确的取值？有些算法在遍历数据结构时，会把一些维护其工作所用的信息记录在数据结构中，对于这些算法来说，尤其应该进行这样的检查。
- 算法所获得的资源，如文件句柄及锁，有没有正确地释放？

同样的方法也可以用在更为高层的操作与配置环境中。例如，如果要验证SQL语句是否正确地构建了某张表格，我们可以查看它所扫描的那些表格及视图，并且看看它构建出来的那张表格是什么样子。如果要判断基于文件的处理流程是否正确，我们可以检查其输入文件与输出文件。如果要调试某个构建在Web服务上面的操作，我们可以检查其中每项Web服务的输入与输出。如果要排解整个数据中心的故障，我们可以检查其中每个元素所需要的及所提供的机制是否正确，其中包括网络连接、DNS、共享存储、数据库以及中间件等。在这些情况下，我们都必须亲自验证（verify），而不能想当然地接受假设（assume）。

要点

- 仔细检查例程的前置条件与后置条件。

第4条：从具体问题入手向上追查bug，或从高层程序入手向下追查bug

要想确定问题的来源，通常有两种办法。一种是从问题的具体表现入手，向上追查其来源，还有一种是从应用程序或系统的顶层入手，逐步向下探查，直至找到其根源。对于某种类型的问题来说，其中一种方法的效果通常要比另一种更好，但是如果你在采用某个方法时遇到了困境，那么不妨试试另一个方法。

如果问题表现得很明确，那我们就应该从发生问题的地方入手，向上追查bug。这可以分成三种情况。

第一种情况是**程序崩溃**。在这种情况下，为了便于排查问题，我们通常可以考虑用调试器来运行程序，也可以在它崩溃的时候把调试器连接到程序上面，或取得内存转储

信息（参见第 35 条）。我们要检查各变量在程序崩溃时的取值，看看有没有 null 值、损坏的值或未初始化的值，这些都有可能是引发崩溃的原因。对于某些系统来说，我们可以通过 0xBAADF00D（代表 bad food）这样的特殊字节值来找出尚未初始化的变量。维基百科的 Magic Number 词条列出了很多这样的特殊值。找到了取值不正确的变量之后，就应该设法查出导致此现象的原因，为此，我们可以试着在程序发生崩溃的这个例程之内探查，也可以沿着调用栈向上寻找不正确的参数或与崩溃有关的其他因素（参见第 3 条和第 32 条）。

如果这些办法找不到原因，那么可以用调试器来多次调试程序，每次都在有可能发生运算错误的地方附近设置断点。像这样反复地设置断点并沿着调用序列上移，或许可以帮助我们查出问题的原因。

第二种情况是**程序冻结**（freeze），这与程序崩溃有所区别，因此我们向上排查所用的办法也稍有不同。我们可以用调试器来运行程序，或将其连接到程序上面，然后用相应的调试器命令来中断其执行过程（参见第 30 条），或使程序生成内存转储信息（参见第 35 条）。有时你会发现，程序所执行的某些代码，并不是该程序自身的代码，而是某个程序库中的例程。无论中断发生在何处，我们都可以沿着调用栈向上排查，以便确定导致程序冻结的那个循环。检查该循环的终止条件，并试着找出它永远无法得到满足的原因。

第三种情况是程序在出现问题时发出了**错误消息**，此时我们首先应该在程序的源代码里找到消息文本的位置。这可以通过 fgrep-r 命令轻松地实现（参见第 22 条），该命令能够在任意深度和复杂度的目录结构中快速定位到待搜索的词句。对于当今很多本地化的软件来说，该命令所定位到的内容，通常并不是发出错误消息的那行代码，而是与错误消息相对应的那个字符串资源文件。例如，如果你住在讲西班牙语的地方，并且正在调试 Inkscape 绘图程序中与“ Ha ocurrido un error al procesar el archivo XCF”错误消息有关的问题，那么用 fgrep-r 命令搜索 Inkscape 的源代码之后，它就可能会把你引向名为 es.po 的西班牙语字符串翻译文件：

```
#: ../share/extensions/gimp_xcf.py:43
msgid "An error occurred while processing the XCF file."
msgstr "Ha ocurrido un error al procesar el archivo XCF."
```

从字符串翻译文件中，我们可以得知与错误消息相对应的源码位置（对于上例来说，就是 share/extensions/gimp_xcf.py 文件的第 43 行）。然后，我们可以在发出错误消

息的源代码这里设置断点，或在它之前插入 log 语句，以检查程序运行到此处所发生的问题。在这种情况下，我们有可能也要后退几行或沿着调用栈向上回溯几层，才能够找到问题的根源。如果你要搜索的是非 ASCII 文本，那么请确保命令行的 locale（区域）设置与源代码所用的文本编码（如 UTF-8）相符。

如果无法确定与故障有关的代码到底在哪里，那我们就应该从顶层系统开始，逐步向下查找故障原因。从定义上来说，这种故障通常属于系统的涌现属性（emergent property），也就是无法与某个具体部分直接对应起来的属性，例如，性能问题（软件占用的内存过多或响应时间过长）、安全问题（Web 应用程序的页面遭到破坏）以及可靠性问题（软件无法提供预期的 Web 服务）等。

要想由上而下地排查错误，我们需要把整个程序分成多个部分，然后分别判断每一部分在引发当前故障的各种因素中可能占多大的比例。对于性能问题来说，常见的办法是做 profile（性能分析），也就是用一些工具和程序库来帮助我们寻找占用 CPU 资源及内存过多的例程。对于安全问题来说，我们要检查代码中有哪些地方可能出现常见的安全漏洞，如缓冲区溢出、代码注入以及跨站脚本攻击等。面对这类问题，我们也可以求助于一些代码分析工具（参见第 51 条）。最后，对于无法提供 Web 服务的问题来说，我们需要审视内部和外部的各种依赖关系，看看它们有没有在正常地运作。

要点

- 如果能够明确指出故障的原因，那么应该从下往上查找错误，例如，在程序崩溃、程序冻结以及程序发出错误消息等情况下，就应该如此。
- 如果故障的原因很难锁定，那么应该从上往下查找错误，例如，在遇到性能问题、安全问题以及可靠性问题的时候，就应该如此。

第 5 条：在能够正常运作的系统与发生故障的系统之间寻找差别

我们通常都能够同时访问这样两个系统，其中一个是发生故障的系统，另一个是与之相似但却可以正常运行的系统。当我们实现了某项新功能、更新了某些工具或基础

组件，或是把系统部署在某个新的平台上面时，就可能会遇到新系统无法正常运行的问题，此时如果旧系统依然正常，那么我们通常可以通过寻找（下面就会讲到如何寻找）或尽量缩小（参见第 45 条）新旧两个系统之间的差别来锁定问题的原因。

之所以能根据新旧系统间的差距来进行调试，其原因在于：尽管各人所经历的问题有所不同，但计算机的底层运作方式却是十分确定的，也就是说，同样的输入会产生同样的输出。因此，只要能够深入故障系统中，并对其进行足够的探查，我们就迟早能够找到相关的 bug，从而揭示出该系统为什么会在行为上与正常系统有所不同。

其实有很多时候，系统的故障原因都会非常明确地出现在你面前，只要你肯打开程序的日志文件（参见第 56 条），就有可能发现里面有一条消息告诉你，clients.conf 这个配置文件有错误：

```
clients.conf: syntax error in line 92
```

在另外一些情况下，错误的原因可能会隐藏得比较深，此时你必须提升系统日志的详细程度（verbosity），才能把它暴露出来。

如果系统没有提供足够详细的日志机制，那我们就需要用追踪工具来梳理其运行时的行为。除了 DTrace 和 SystemTap 等通用的工具，还有一些专门的工具可以用来追踪对操作系统的调用（strace、truss、Procmon）、对动态链接库的调用（ltrace、Procmon）、网络包（tcpdump、Wireshark）以及 SQL 数据库调用（参见第 58 条）。有很多 Unix 应用程序（如 R Project）是借助复杂的 shell 脚本来启动的，因此可能会以极其隐晦的方式出错。针对这样的错误，在大多数情况下，我们都可以通过给相应 shell 传入 -x 选项的办法来进行追踪，这样得到的数据通常很庞大，所幸现在的系统都有很大的容量能够存放这两份日志（以其中一份表示那个可以正常运作的系统，另一份表示出现了故障的系统），而且都有很强的 CPU 能够对其进行处理与比较。

就系统的操作环境而言，我们应该尽量确保这两个系统拥有相似的环境，因为这样能够更加方便地对比日志文件或追踪信息，有时甚至可以直接找到造成 bug 的原因。我们可以先从一些较为明显的部分入手，例如，程序的输入以及命令行参数等。与早前所说的原则一样，我们也要亲自进行验证，而不能想当然地接受假设。例如，应该在两个系统的输入文件之间进行对比，如果它们都比较庞大并且离得比较远，那可以考虑对比它们的 MD5 校验和。

然后，我们应该把重点放在代码上。首先对源代码进行对比，我们可能要挖得深一

些才能找到 bug 所在的地方。可以通过 ldd 命令（适用于 Unix 系统）或是带有 /dependents 选项的 dumpbin 命令（适用于 Visual Studio）来查看与每个可执行文件有关的动态程序库，并通过 nm 命令（适用于 Unix 系统）、带有 /exports/imports 选项的 dumpbin 命令（适用于 Visual Studio）或 javap 命令（适用于以 Java 语言开发出来的程序）来查看程序所定义和使用的符号。如果你确信问题肯定出现在代码中，但又看不出明显的差别，那么可能就要往更深的层次去探查了，也就是需要对比由编译器所生成的汇编代码（参见第 37 条）。

然而在进行更深层次的探查之前，应该先考虑一下有没有其他因素会影响程序的执行情况，环境变量就是这样一个容易忽视的因素，即便是没有特权的用户，也依然可以通过设置环境变量来破坏程序的正常执行。另一个因素是操作系统。与运行着正常程序的那个操作系统相比，故障程序所在的这个操作系统，可能新了 10 年或是旧了 10 年。此外，也要考虑编译器、开发框架、第三方链接库、浏览器、应用程序服务器、数据库系统以及其他一些中间件。至于怎样在这么多的因素中确定问题的根源，则是我们接下来要讲的话题。

大多数情况下，我们都是在一堆干草里面找一根针（大海捞针），因此应该尽量使这堆干草变得小一些，于是，就要花时间来构造一个既能体现 bug，又最为简单的测试用例（参见第 10 条）。（另外一种办法是把要找的针变大一些，也就是命令这个有 bug 的程序输出更多的信息，然而这种做法很少能起到比较好的效果。）简明的测试用例可以缩短日志文件与追踪信息的长度并减少处理时间，从而令调试工作变得更加轻松。要想有条理地简化测试用例，我们可以在确保能够重现 bug 的前提下，逐渐删除用例中的元素或系统中的配置选项，直到删至最简。

如果正常系统和故障系统的区别位于源代码中，那么有一种很实用的办法，就是对这两个版本之间的历次修改进行二分搜索（binary search），以确定问题所在。例如，如果正常系统的版本号是 100，而故障系统的版本号是 132，那我们首先测试 116 版的程序是否正常，如果 116 版正常，那就判断它与 132 版之间的中点，也就是 124 版是否正常，如果 116 版有错，则判断它与 100 版之间的中点，也就是 108 版是否正常，并依此类推。每次修改完程序之后，我们都应该把代码单独提交到版本控制系统里面，这样做的好处之一，就是使得我们能够进行二分搜索。某些版本控制系统提供了可以自动执行搜索的命令，例如，Git 就提供了 git bisect 命令（参见第 26 条）。

还有一个很有效的办法，是用 Unix 工具对比两份日志文件（参见第 56 条），以找

出其中与 bug 有关的区别。我们在这种情况下所使用的工具，是 diff 命令，它可以显示出两份文件的不同之处。然而日志文件经常会在无关紧要的地方表现出差别，这会把那些与 bug 真正有关的差别给掩盖掉，于是，我们可以考虑用各种办法来过滤干扰因素。例如，如果每一行开头的几个字段，都是时间戳与进程 ID 等信息，那我们就可以用 cut 或 awk 命令来把这些大同小异的信息裁掉。下面这条命令可以对 Unix 系统的 messages 日志文件进行裁切，它会从每一行的第 4 个字段开始显示其内容：

```
cut -d ' ' -f 4- /var/log/messages
```

只把你感兴趣的那些事件选出来就可以了，例如，如果你只对打开的文件感兴趣，那么可以用 grep'open(' 这样的命令来进行筛选。你也可以用 grep-v gettimeofday 等命令来把对自己有干扰的文本行过滤掉（例如，在 Java 程序里面，会有成千上万次与获取系统时间有关的调用）。此外，还可以在 sed 命令中指定适当的正则表达式，以便把文本行中自己不感兴趣的那一部分裁掉。

最后再讲一个高级的实用技巧：如果两份文件各自的排序方式无法使 diff 命令给出有效的对比结果，那我们可以把感兴趣的字段提取出来，对其进行排序，然后用 comm 工具在排好顺序的两个集合中找寻不同的元素。例如，如果我们想对比 t1 和 t2 这两份追踪信息，以找出有哪些文件只出现于 t1 中，那么可以在 Unix 的 Bash shell 中输入下列命令，它会在包含字符串 open(的那些文本行里面提取表示文件名的第二个字段，并在提取出来的这两个集合之间寻找差别：

```
comm -23 <(awk '/open\(/{print $2}' t1 | sort) \
      <(awk '/open\(/{print $2}' t2 | sort)
```

两对小括号里面的那两个元素，会分别生成两份有序列表，列表中的每一项都是一个传给 open 的文件名，而 comm 命令（这个命令用来在两份列表之间寻找共同的元素）则以这两份列表为输入值，并把只出现在第一份列表中的内容列出来。

要点

- 在能够正常运作的系统与出现故障的系统之间对比，找出行为上的区别，以求发现故障的原因。
- 影响系统行为的所有因素都要考虑到，包括代码、输入、调用时的参数、环境变量、服务以及动态链接库。

第 6 条：使用软件自身的调试机制

程序是一种很复杂的东西，因此它们通常都包含内置的调试机制。（至于怎样给自己正在开发的软件里面添加这样的机制，请参见第 40 条。）这种机制有很多好处，其中包括：

- 我们可以通过禁用后台执行或多线程执行等特性来简化程序的调试工作。
- 我们可以有选择地执行其中某一部分功能，以便通过测试用例来精确地再现相关的故障。
- 程序可以给我们提供与性能有关的报表及其他信息。
- 程序可以把更多的信息记录在日志文件中。

因此，我们应该花一些时间，看看自己要调试的这款软件内置了哪些调试机制。想要了解这些机制，我们可以在程序文档和源代码里面搜索 debug 这个词，这样或许就能找到与开启调试模式有关的命令行选项、配置文件中的条目、构建选项、信号（适用于 Unix 系统）、注册表中的条目（适用于 Windows 系统）或可以在命令行界面中执行的命令。

一般来说，开启了调试选项之后，程序所输出的信息会更为详细，于是，它的操作就会变得更加明确，有时甚至还会变得更加简单。（但糟糕的是，开启了这些选项之后，有些 bug 也会随之消失。）由于程序的日志文件所记录的信息要比没有开启调试模式的时候更多，因此，我们可以通过这些内容来探究故障背后的原因（参见第 56 条）。下面举几个例子。

有些程序很容易就能通过一个命令选项来启用其调试机制，启用了之后，它在执行操作时就会给出更为详细的信息。例如，Unix shell 就提供了 -x 选项，用来详细展示该 shell 所执行的命令。在调试一些比较微妙的文本替换问题时，这是很有用的。下面这段代码是一个可以用 shell 来执行的循环：

```
find-git-repos |
while read repo ; do
  data=$(echo $repo | sed 's/repos/data/')
  # Skip if done
  test -r $data.out && continue
  # Obtain time series
  qtimeseries $(pwd)/$repo >$data.out 2>$data.err
done
```

把上述代码放在脚本文件里，然后用开启了命令追踪机制的 shell 来执行它：

```
sh -x script.sh
```

执行的时候，shell 会产生下面这样的信息，这些信息详细展示了命令所输出的内容以及替换之后的变量：

```
+ read repo
+ echo repos/mysql-server
+ sed s/repos/data/
+ data=data/mysql-server
+ test -r data/mysql-server.out
+ continue
```

为了能够更加顺利地进行调试，我们在启动程序的时候，通常可以把多个选项结合起来使用。例如，我们要排查 ssh 连接失败的原因。假如修改全局的 sshd 配置文件或钥匙，那么就可能会使其他人无法进行连接，因此，我们可以考虑用 -f 选项来启动 sshd，以便指定一份定制好的配置文件，并且用 -p 选项来指定一个与默认端口不同的端口（请注意，客户端连接服务器时，也必须使用这个端口），此外，还要加上 -d（调试）选项，使得进程运行在前台，并把调试消息显示在终端窗口中。我们分别在服务器端与客户端采用下面这两条命令来启动服务并进行连接：

```
# Command run on the server side
sudo /usr/sbin/sshd -f ./sshd_config -d -p 1234
# Command run on the client side
ssh -p 1234 server.example.com
```

这些命令运行起来之后，会产生图 1.1 这样的调试信息，其中的最后一条信息揭示了连接失败的原因。

调试机制也可以帮助我们排查性能方面的问题。例如，有下面这两条 SQL 查询语句：

```
select count(*) from commits where au_id = 1;
select count(*) from commits where
  created_at = '2012-08-01 16:25:36';
```

第一条语句不到 10 毫秒就可以完成，但是第二条语句却用了 3 分钟还没有执行完。我们现在用 SQL 的 explain 机制来解释第二条语句耗时过长的原因（对于本例来说，这个原因当然是显而易见的）：

```
explain select count(*) from commits where au_id = 1;
explain select count(*) from commits where
  created_at = '2012-08-01 16:25:36';
```

从图 1.2 中的输出信息可以看到：由于第一条查询语句使用了索引，因此只需要扫描 21 个数据行就可以找出匹配的结果，而第二条查询语句没有使用索引，因此必须把

非卖品，仅供非商业用途或交流学习使用

表格里面的两亿两千二百多万个数据行全都扫描一遍，才能从中查出4个匹配的结果。

```
debug1: sshd version OpenSSH_6.6.1p1_hpn13v11 FreeBSD-20140420,
OpenSSL 1.0.1p-freebsd 9 Jul 2015
debug1: read PEM private key done: type RSA
[...]
Server listening on :: port 1234.
debug1: Server will not fork when running in debugging mode.
debug1: rexec start in 5 out 5 newsock 5 pipe -1 sock 8
debug1: inetd sockets after dupping: 3, 3
Connection from 10.212.168.34 port 57864
debug1: Client protocol version 2.0; client software version
OpenSSH_6.7p1 Debian-5
[...]
debug1: trying public key file /home/dds/.ssh/authorized_keys
debug1: fd 4 clearing O_NONBLOCK
Authentication refused: bad ownership or modes for directory
/home/dds/.ssh
```

图 1.1 由开启了调试模式的 ssh 守护进程所输出的调试信息

```
+----+-------------+---------+------+---------------+-------+
| id | select_type | table   | type | possible_keys | key   |
+----+-------------+---------+------+---------------+-------+
|  1 | SIMPLE      | commits | ref  | au_id         | au_id |
+----+-------------+---------+------+---------------+-------+
+---------+-------+------+--------------------------+
| key_len | ref   | rows | Extra                    |
+---------+-------+------+--------------------------+
| 5       | const |   21 | Using where; Using index |
+---------+-------+------+--------------------------+
```

```
+----+-------------+---------+------+---------------+------+
| id | select_type | table   | type | possible_keys | key  |
+----+-------------+---------+------+---------------+------+
|  1 | SIMPLE      | commits | ALL  | NULL          | NULL |
+----+-------------+---------+------+---------------+------+
+---------+------+-----------+-------------+
| key_len | ref  | rows      | Extra       |
+---------+------+-----------+-------------+
| NULL    | NULL | 222526953 | Using where |
+---------+------+-----------+-------------+
```

图 1.2 由 SQL 的 explain 机制所输出的信息，上半部分对应于使用了索引的查询语句，下半部分对应于未使用索引的查询语句

还有一种调试机制，使得我们可以精确地定位到某一种情况。例如，某台主机正在忙碌地投递上千条电子邮件消息，其中有一条消息无法投递。我们要想知道无法投

非卖品，仅供非商业用途或交流学习使用

递的原因，可以调用 Postfix 的 sendmail 命令，同时以 -v 选项来开启详细输出模式，并以 -M 选项来指定那条消息的标识符。

```
sudo sendmail -v -M 1ZkIDm-0006BH-0X
```

执行上述命令之后，可以看到类似下方所示的输出信息。与刚才那个例子类似，在本例中，问题的原因也写在最后一行。

```
delivering 1ZkIDm-0006BH-0X
LOG: MAIN
  Unfrozen by forced delivery
R: smarthost for root@example.com
T: remote_smtp_smarthost for root@example.com
Connecting to blade-a1-vm-smtp.servers.example.com
[10.251.255.107]:25 ... connected
  SMTP<< 220 blade-a1-vm-smtp.servers.example.com ESMTP
  SMTP>> EHLO parrot.example.com
  SMTP<< 250-blade-a1-vm-smtp.servers.example.com Ok.
        250-AUTH LOGIN
        250-STARTTLS
        250-XEXDATA
        250-XSECURITY=NONE,STARTTLS
        250 DSN
  SMTP>> STARTTLS
  SMTP<< 220 Ok
  SMTP>> EHLO parrot.example.com
  SMTP<< 250-blade-a1-vm-smtp.servers.example.com Ok.
        250-AUTH LOGIN
        250-XEXDATA
        250-XSECURITY=NONE,STARTTLS
        250 DSN
  SMTP>> MAIL FROM:<> SIZE=2568
  SMTP>> RCPT TO:<root@example.com>
  SMTP>> DATA
  SMTP<< 535 Authentication required.
```

要点

- 找出你正在调试的这款软件所支持的调试机制，并以此来排查你所遇到的问题。

第 7 条：试着用多种工具构建软件，并将其放在不同的环境下执行

有时我们可以通过改变环境来锁定一些难以捕获的 bug。例如，我们可以用另外一

非卖品，仅供非商业用途或交流学习使用

款编译器来构建这个软件，也可以切换到其他的运行时解释器、虚拟机、中间件、操作系统或 CPU 架构上。由于那些环境可能会更加严格地检查输入数据，或能通过其结构来凸现程序中的错误（参见第 17 条），因此可以帮助我们发现原来很难找到的一些 bug。如果程序不够稳定、总是发生无法重现的崩溃问题，或移植起来不太顺利，那就应该试着把它放在另外一种环境下进行测试，这使得我们能够使用更为先进的调试工具，例如，一款图形界面很漂亮的调试器或 dtrace 等（参见第 58 条）。

在其他操作系统里面编译或运行软件，可以把我们对 API 的用法所做的错误假设暴露出来。例如，由于某些 C 及 C++ 头文件声明了很多不一定会用到的实体，因此我们总是认为只要把这些头文件包含进来就够了，这可能导致我们忘记将另外一个必需的头文件包含进来，从而使客户遇到移植方面的问题。此外，某些 API 的实现方式在各种操作系统之间会有很大的区别，例如，Solaris、FreeBSD 及 GNU/Linux 系统会采用不同的方式来实现 C 程序库，而桌面版与移动版的 Windows API，目前也是基于不同的代码库来实现的。请注意，那些在底层采用 C 程序库和 API 的解释型语言也会受到影响，如 JavaScript、Lua、Perl、Python 或 Ruby 等。

对于 C 和 C++ 这样较为接近硬件的语言来说，底层的处理器架构会对程序的行为造成影响。过去几十年间，Intel x86 架构与 ARM 架构分别成为桌面市场与移动市场的主流，于是，那些在字节序上面（SPARC、PowerPC）或是在对空指针的解引用上面（VAX）偏离主流的架构，就变得不那么流行了。然而，x86 架构与 ARM 架构在处理未对齐的内存访问及内存布局时依然有所区别。例如，如果在奇数内存地址处访问两字节的值，那么就有可能令某些 ARM 架构的 CPU 出错，或令 CPU 表现出非原子的（non-atomic）行为。在其他架构上面进行未对齐的内存访问，可能会严重影响程序的性能。此外，结构体的大小，以及其中各成员距离结构体开始处的偏移量，在这两种架构中也是有区别的，对于老版本的编译器来说，这种区别尤其突出。还有一个更为重要的问题在于：当你把代码从 32 位架构移植到 64 位架构，或是从一个操作系统移植到另一个操作系统的时候，长整数（long）及指针值等原始类型所占据的大小可能也会有所改变。下面这个程序可以显示出五种原始类型所占据的字节数。

```
#include <stdio.h>

int
main()
{
        printf("S=%zu I=%zu L=%zu LL=%zu P=%zu\n", sizeof(short),
```

非卖品，仅供非商业用途或交流学习使用

```
        sizeof(int), sizeof(long),
        sizeof(long long), sizeof(char *));
}
```

在较为典型的几种环境之下，上述程序所给出的结果分别是：

```
Windows, Microsoft C/C++ 80x86   S=2 I=4 L=4 LL=8 P=4
Windows, Microsoft C/C++ x64     S=2 I=4 L=4 LL=8 P=8
GNU/Linux, GCC, x86_64           S=2 I=4 L=8 LL=8 P=8
GNU/Linux, GCC, armv6l           S=2 I=4 L=4 LL=8 P=4
OS X, LLVM, x86_64               S=2 I=4 L=8 LL=8 P=8
```

由此可见，把软件放在其他架构或操作系统中运行，可以帮助我们对其进行调试，并检测出移植方面的问题。

对于移动平台来说，各种设备之间的差距要比桌面平台更大，它们不仅在操作系统的版本方面有所不同（大多数手机与平板厂商都会对原始的 Android 系统进行修改，并把改版后的系统安装在设备上），而且在硬件方面也有着相当大的区别，如屏幕分辨率、操作界面、内存及处理器等。这使得我们在开发移动软件时，更有必要将其放在各种不同的设备上进行调试，为此，很多移动 app 的开发团队都有许多种移动设备。

在其他执行环境中调试代码，主要有三种方式：

1. 在工作站安装虚拟机软件，并且用虚拟机来运行各种不同的操作系统。这种办法还有一个好处，就是可以保留各种执行环境的原始镜像，我们对虚拟机进行配置时，是在这个原始镜像的基础之上进行修改的，如果有必要，我们可以把虚拟机恢复到原始状态。

2. 使用小型的廉价计算机。如果你主要面对的是 x86 架构，但同时又想尝试 ARM CPU，那么最简单的办法就是使用 Raspberry Pi（树莓派）。这是一种基于 ARM 的微型设备，能够运行很多种流行的操作系统。它可以连接网线，或通过 Wi-Fi 上网。我们可以在这种设备上尝试 GNU/Linux 的开发环境，这对于主要在 Windows 或 OS X 系统上调试代码的人来说是很有帮助的。此外，如果你平常使用的是 Windows 系统，那么可以买一台 Mac mini，这样就能够轻松地切换到 OS X 开发环境了。

3. 租用基于云端的主机，并在上面运行你想使用的操作系统。

要想用各种不同的编译器与运行时环境来调试代码，我们固然可以安装新的操作系统或使用新的设备，但除此之外还有一种办法，那就是设法使自己这台工作计算机上面的开发环境变得更加丰富。这样，我们就可以看到由其他开发工具所给出的错误与警告信息，并且可以用那些工具对代码中的某些方面进行更为严格的检查，看看它们有没有

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

遵从相关的规范。与使用静态分析工具（参见第 51 条）所带来的好处类似，同时使用多种编译器，可以使我们发现更多的问题，这其中既包括移植方面的问题，也包括逻辑方面的问题。例如，如果某一款编译器对代码检查得比较宽松，而另外一款编译器检查得比较严格，那么就可以揭示出移植方面的问题；如果某一款编译器不对某个逻辑问题发出警告，而另外一款编译器对此发出了警告，那么就可以揭示出逻辑方面的问题。只要是符合语法的代码，编译器基本上都可以把它编译成可执行的文件，但是它们有时不太能够检查出对编程语言的误用，例如，即便代码所引入的头文件里面声明了一些没有公开发布的元素，有些编译器也依然会直接编译通过，而不会指出相关的问题，为此，我们应该多用几种编译器进行编译，以便把这方面的问题暴露出来。为了使开发环境变得更加丰富，我们在调试软件的过程中，不仅要使用主流的工具，而且还要同时安装并使用一些替代产品。下面给出几条建议：

- 开发 .NET Framework 程序的时候，不仅要使用 Microsoft 的工具与环境，而且还要同时使用 Mono。
- 用 Ada、C、C++、Objective C 或其他相关的语言来编写程序时，要同时使用 LLVM 与 GCC 这两种编译器。
- 开发 Java 程序时，要同时使用 OpenJDK（或由 Oracle 公司基于相同的代码库所提供的 JDK）及 GNU Classpath 这两种开发包。此外也要注意把程序放在一种以上的 Java 运行时环境里面执行。
- 开发 Ruby 程序时，不仅要使用作为参考实现的 CRuby 来进行开发，而且还要尝试其他 VM，如 JRuby、Rubinius 及 mruby。

还有一种更为大胆的做法，那就是把程序里面的一部分代码改用另外一种语言来重新实现。如果你要调试的是个比较麻烦的算法，那么这种做法就很有用处。最为典型的情况是：你起初用了一种较为低级的语言来实现这个算法，然后发现这种实现方式无法运作，于是，你考虑采用 Python、R、Ruby、Haskell 或 Unix shell 等更为高级的语言来重新实现它。在实现过程中，你应该使用这些语言所提供的高级特性，例如，可以对集合、管道与过滤器进行操作，并使用高阶函数（higher-order function）等，这样能够帮你实现出一个可以正常运作的算法。当你迅速查明算法的设计问题并将实现中的错误加以修复之后，如果觉得这种新的实现方式在性能上无法满足要求，那么可以回过头去，用原来的语言或某种较为接近 CPU 的语言，把算法再重新实现一遍，并采用各种对比式的技术进行调试（参见第 5 条），使其能够正常运作。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

要点

- 用多种编译工具来构建软件，并将其放在各种平台中执行，可以给调试工作提供很多有价值的思路。
- 如果遇到了一个很难调试的算法，那么可以考虑改用高级语言将其重新实现一遍。

第 8 条：把工作焦点放在最为重要的问题上

许多大型软件系统都含有数量极其众多的 bug（有一些是已知的 bug，还有一些则尚未发现）。要想高效地进行调试，就必须把应该受到关注的 bug 与可以忽略的 bug 明智地区分开。这样做不是为了单纯地缩减事务清单中的未决事务，而是为了帮助我们开发出稳定、易用、可维护而且效率较高的软件，毕竟这才是公司给我们支付薪水的原因。为此，我们要通过事务追踪系统来设定各项事务的优先级（参见第 1 条），从而使自己能够把工作重心汇聚在优先级较高的那些事务上，并把优先级较低的事务忽略掉。下面给出一些与排定事务优先级有关的建议。

下列几类问题应该赋予较高的优先级。

- **数据丢失**：数据之所以会丢失，可能是因为本身受到了损坏，也可能因为程序在可用性方面遇到了问题。用户把他们的数据托付给你的软件，你就应该维系这份信任感。如果数据丢失，那你就违背了他们的信任，要想再重新获得其信任，是很困难的。
- **数据安全**：这种问题可能会影响软件数据的保密性与完整性，也可能会影响软件所在系统的完整性，或软件所提供服务的可用性。此类问题通常是由于恶意人士对软件进行攻击而引发的，它会给公司带来巨大的资金和名誉损失，此外，还会引来监管机构的注意，或招致勒索。因此，安全问题必须尽快解决。
- **服务的可用性降低**：如果软件是用来提供某种服务的，那么当这项服务无法使用时，公司就会遭受资金损失（有时甚至是以百万元为单位的损失），此外，还会令公司失去信誉、迫使经理在半夜愤怒地打电话给你，而且也会使服务人员忙得不可开交。这些都是应该尽量避免的事情。
- **使用安全**：此类问题可能导致用户伤亡，使财产丢失或受损，或令环境遭到破坏。前面那几类问题所造成的后果，同样有可能出现在这一类问题上面。如果

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

软件可能会在这一方面发生故障，那就应该用一种比这份清单更加严谨的流程来指导你的行动。

- **程序崩溃或冻结**：这可能导致数据丢失或服务下线，而且此类问题可能与底层的安全问题有关。在程序崩溃或失去响应之后，我们通常可以通过事后分析技术（参见第35条）来对其进行调试。这种问题自然不应该赋予较低的优先级。
- **代码质量**（code hygiene）：我们要把编译器所给出的警告信息和断言失败信息，以及未处理的异常与内存泄漏等问题清理干净。总之，由于各种低质量的代码会滋生并掩藏一些严重的bug，因此，我们不能允许这类问题继续存在并积累下去（参见第20条）。

下面几类问题的优先级可以设置得低一些。这并不是说它们本身不重要或不值得关注，而是说为了解决更为紧迫的问题，我们可以把这些问题先放在一边。

- **对遗留事物的支持**：能够支持过时的硬件、API或文件格式，这固然是好的，然而从商业角度来看，这样做不会有太大意义，因为使用这些东西的人已经越来越少了。
- **向后兼容**：这一类问题的优先级没有较为明确的结论。如果你在软件的发展过程中，总是把原来的用户抛开不管，那就会失去顾客对你的信赖。有些公司，如Nikon，通过对向后兼容性的维护而获得了良好的口碑，他们的新产品能够与很多代之前的旧产品相兼容，例如，当前的高端Nikon相机，依然那可以使用20世纪70年代的Nikkor（尼克尔）镜头。与之相对，某些成功的软件公司以其决绝的做法而知名，他们毫不犹豫地放弃对旧软件与旧服务的支持。有时我们确实应该停止对旧特性的支持，以便把精力更好地放在未来的发展上面。
- **美观问题**：这些问题很难处理得十分恰当，而且经常容易遭人忽视。例如，就算弹出式帮助信息没有完整地显示出来，客户也不太可能因为这个而抛弃你的产品，可是当你要修复这个小问题的时候，却会发现它处理起来相当麻烦，因为你得根据屏幕的分辨率设置来动态地调整帮助面板的尺寸。
- **已经有了临时解决方案的问题**：有些bug调试起来比较复杂，为了避免在这些问题上面消耗时间，我们可以先给用户提供一种临时的解决方案，以便暂时绕过这些问题。例如，笔者在打开自己家的电视之后，想试着用电视的遥控器去操作媒体播放机，然而却看到了一条提示信息：“请再试一次”。我怀疑这个小问题修复起来相当麻烦，于是厂商就先给出了这个权宜的办法。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

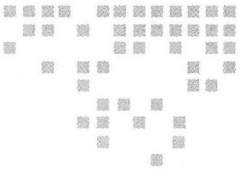
- **很少有人会用到的特性**：当软件中的某一个较为奇怪，而且很少有人用到的特性出了问题时，我们与其花时间去解决这个问题，还不如直接把该特性删掉（并把由此引发的一些小状况处理好）。通过收集软件的使用数据，我们可以更为容易地确定出这些罕用的特性，并据此做出决策。

请注意，如果你决定忽略某个优先级比较低的问题，那就应该把这个态度明确地表达出来。你可以在事务追踪系统里面表明自己“不打算解决”该问题，并将其关闭。这样做可以把自己的决策记录下来，使得其他人以后不会再提出类似的问题，从而减轻管理方面的工作量。

要点

- 并不是所有的问题都值得解决。
- 修复优先级较低的问题可能会耽误你的时间，使你无法拿出更多时间去处理那些更为紧迫的事务。

非卖品，仅供非商业用途或交流学习使用



第 2 章 Chapter 2

通用的方法与做法

调试故障所用的办法，通常要根据其背后的技术与开发平台而定，不过，有一些办法是可以适用于各种情况的。

第 9 条：相信自己能够把问题调试好

软件通常是极其复杂的。机械表的移动机制，仅由 100 多个部件组成；而整个房屋中的各种器械，其部件总数也只是简单组件的几倍而已。这与典型的软件系统有很大区别，后者很容易就包含成千上万行复杂的代码。我们可以在这两个领域中各举一个较为精密的例子来做比较：A380 客机有 400 万个物理组件，而 Linux 内核的代码行数则是 900 万。因此，我们必须在思想上做好充分的准备，才能应对如此复杂的软件。

首先，你要确信自己一定能够找到问题并将其修复。你的心理状态会对调试的结果造成影响，专家们把这叫做“感受到的挑战与自身技能之间的一场对抗”。如果你根本就不相信自己能够克服这个问题，那你的思维就会徘徊不前，甚至想要干脆放弃。在这种情况下，你解决不了实质的问题，而是会盲目地打补丁，以掩盖由该问题所引发的各种症状，这样做对代码是有害的。我们必须记住这一点。

如果问题是可以重现的，那么毫无疑问，你肯定能解决它（而且通常可以按照本书所给出的建议来解决）。如果问题不能重现，那么有一些办法可以令其变得能够重现。

非卖品，仅供非商业用途或交流学习使用

在调试的时候，有两个重要的“朋友”可以帮助我们：一个是对数据的访问权，它使我们能够访问到自己所需的全部数据；另一个是功能强大的计算机，它使我们能够对这些数据进行处理。我们可以检查程序发生问题时所表现出来的状况，以及程序的日志和程序的源代码，有时甚至可以检查机器指令，此外，我们也可以在软件栈的任意位置添加详细的日志语句（或者说至少添加一些监测探针）。然后，就可以用工具或较短的脚本来筛选这些数据，并从中找到问题的根源。这是一种综合能力，具备这种能力的人可以在较大的范围内及任意的深度上面进行搜索，从而完成调试工作，这个过程会给人一种独特的满足感。

为了能够高效地进行调试，你还必须留出充足的时间。调试是一项对人要求很高的工作，它比编程更为复杂，因为我们不仅要明白程序的逻辑，而且还要理解其背后的效果（这通常指的是较为底层的效果）。此外，我们还必须把环境、断点、日志记录、各种窗口以及测试用例设置好，以便能够高效地重现问题。不要在还没有解决 bug 的时候就停手，否则前面所花的时间就全都白费了，即便要停手，也应该在准确理解了接下来所应采取的措施之后再停止。

要想调试复杂的问题，就必须在没有干扰的状态下工作。人脑需要经过一段时间之后才能进入心流（flow）状态，在这种状态下，我们会完全沉浸于自己正在做的事情中。心流这个概念由 Mihály Csíkszentmihályi 提出，根据他的说法，人在处于心流状态时，能够把自身情绪与自己所做的事情相互契合起来，并通过一种成就感来提升自己做事的毅力与效果。我们在调试复杂的系统时会遇到巨大的困难，而心流状态所带来的好处对于解决这些困难会起到关键的作用。弹出的消息、打来的电话、反复的聊天、持续更新的社交网络，或是跑过来求助的同事，都会对自己造成干扰，从而破坏这种心流状态。一旦离开这种状态，就享受不到它所带来的好处了，因此，我们要尽量避免干扰，应该把用不到的应用程序关掉，把电话调到静音模式，可以在显示器上贴一个请勿打扰的标志（如果你有一间自己的办公室，那也可以把牌子挂到门上）。

还有个很有用的办法，就是在遇到难题的时候去睡一觉。研究者发现，人在睡觉的时候，其神经元之间会形成广泛的连接，这些连接会把看似不相关的路径贯通起来。这对于调试工作可以起到很大的帮助作用，它通常可以使我们找到一种打破思维定式的调试策略，从而跳出当前的困局。在看似不相关的路径之间所形成的这种新连接，是在睡眠的过程中搭建起来的，然而要想使这种机制有利于调试工作，我们还必须合理地运用它。也就是说，我们必须先努力地解决问题，使得必要的数据都存留在脑子里，然后才

非卖品，仅供非商业用途或交流学习使用

能在睡眠中找到创新的解法。刚一遇到困难就立刻喝一杯啤酒，然后跑去睡觉，是没有太大作用的。此外还要注意保持充足的睡眠，以便在第二天醒来之后，脑子里与意识有关的那一部分能够听取与潜意识有关的那一部分所给出的建议，从而令工作更有效率。

没有谁会认为调试是一件简单的事情，所以要想高效地调试，就必须有毅力。由于计算机在最底层具有确定性，因此我们最终可以通过深入挖掘把错误的原因隔离出来。在较高一些的层面上，为了提升表达能力与效能，它会引入一些不确定的因素，使得程序的某些行为看起来较为随机（可以思考一下多线程的应用程序是如何运作的）。面对这些与不确定因素有关的错误时，我们要把握住一点：计算机毕竟是一种运行速度很快的可编程机器，它可以运行数量极多的用例，因此，我们最后还是能够把错误原因找出来。由此可见，调试工作之所以会陷入僵局，基本上都是因为缺乏毅力，例如，我们可能没有去写某个测试用例，没有去查看某个日志文件，或是没有去尝试从另外一个角度来研究问题。

最后要注意的是：想成为一名高效的调试工程师，就必须持续地投入精力，去学习环境、工具及相关的知识。只有这样，你才能够在复杂度持续提升的技术工作中保持优势。现在看来，笔者当年在调试时最常犯的错误，就是没有花足够的功夫来把调试工作所需的基础设施搭建好。要想把基础设施搭建好，可能需要把下面四项全都做到：

- ❑ 把健壮的最小测试用例准备好（参见第 10 条）。
- ❑ 对 bug 的重现加以自动化。
- ❑ 用脚本来分析日志文件。
- ❑ 了解 API 或语言特性的实际运作方式。

当我振作起来把精力投入到应该做的事情上面之后，我的调试效率就会急剧提升。一旦进入这种状态，通常可以在几分钟之内查明 bug 的原因。

要点

- ■ 确信问题是可以追查并解决的。
- ■ 给调试工作留出足够的时间。
- ■ 安排好工作环境，使自己不受干扰。
- ■ 遇到难题的时候可以先去睡一觉。

非卖品，仅供非商业用途或交流学习使用

- 不要彻底放弃。
- 投入精力去学习环境、工具及知识。

第 10 条：高效地重现程序中的问题

要想高效地调试程序问题，一个关键的因素就是要能够可靠且方便地重现它。这么说有三个理由。首先，如果我们总是能做到只按一个按钮就可以重现问题，那么自然能够专心地去寻找问题的原因，而不用再浪费时间去研究怎样才能把这个问题重现一遍。第二，如果我们可以方便地重现问题，那么也就能够同样方便地把问题描述出来，以寻求外人的帮助（参见第 2 条）。第三，修复错误之后，我们可以把重现问题所需的步骤执行一遍，如果程序这次没有出现故障，那就证明我们对其所做的修复是正确的。

创建短小的范例或测试用例（test case），以便重现问题，这对于提升调试的效率来说是很有帮助的。我们要遵守最小范例（minimal example）准则，这是一条黄金标准，它要求我们写出可以重现问题的最短范例。还有一条名为 SSCCE 的准则（参见第 1 条），可以称为铂金标准，它要求我们不仅要写出短小的（short）范例，而且还要把它写成自足（self-contained）且正确无误（correct，也就是可以编译、可以运行）的范例（example）。有了这种最小范例之后，我们就不用再花时间去研究那些可以忽略的代码分支了，而且我们所要创建及查看的日志文件与追踪信息，也不会变得过于冗长。此外，这种短小的范例，执行起来也要比那些较长的范例更为迅速，这其中一个重要原因在于调试模式所需的开销相当大。

为了缩短范例的长度，可以考虑自上而下与自下而上这两种办法（参见第 4 条），我们需要根据具体情况来决定应该采用哪种办法。如果代码中的依赖关系比较多，那么自下而上的方法或许会好一些，因为这样可以使我们在刚开始调试的时候，无需面对过多的依赖关系。如果不理解导致问题发生的真正原因，那就应该创建自上而下的测试用例，以帮助我们缩小可能包含错误的代码范围。

自下而上地进行调试时，要先对问题的原因给出一个假设，例如，我们认为是由调用某个 API 所引发的，然后，构建测试用例来演示这个问题。有一次笔者遇到一个用来处理输入文件的程序，这个程序的代码比较复杂，一共有 2 7000 行，而且运行得相当缓慢。在查看了程序所调用的系统操作之后，我猜想问题可能与调用 tellg 函数有关，

非卖品，仅供非商业用途或交流学习使用

在读取文件的时候，这个函数可以返回当前位置在文件流中的偏移量。笔者在运行了下面这个简短的代码片段之后，确认自己的假设是正确的（参见第 58 条），而且这个代码片段还有利于我对该问题的权宜解决方案（也就是使用包装类）进行测试。

```
ifstream in(fname.c_str(), ios::binary);
do {
  (void)in.tellg();
} while ((val = in.get()) != EOF);
```

自上而下地进行调试时，我们要从能够演示问题的场景中逐步移除各种元素，直到不能继续移除。对于这种类型的调试工作来说，二分搜索技术通常是很有帮助的。例如，我们要调试一个无法在浏览器里正常运作的 HTML 文件。首先，可以删去文件的 head 元素。如果问题依然存在，那就删掉 body 元素。删除之后，如果问题消失了，那么就恢复 body 元素，并将其中的一半内容删掉。反复执行此过程，直到我们确定了引发错误的元素。在这个过程中，我们要一直打开编辑器，并且要在进入了错误的路径之后，通过撤销功能返回到正确的路径之上，这样做会极大地提升调试效率。

有了短小的范例之后，我们很容易就能创建出一个自足的范例。这种自足的范例，不应该依赖于外部的程序库、头文件、CSS 文件及 Web 服务等组件，以便使我们可以把它拿到其他地方运行，并将问题重现出来。如果测试用例确实需要使用某些外部元素，那么可以把那些元素与该用例捆绑起来。请注意，我们要使用可移植的形式来引用这些元素，而不要使用绝对的文件路径或固化的 IP 地址。例如，要使用 ../resources/file.css 形式来引用 css 文件，而不能使用 /home/susan/resources/file.css 的形式，要使用 http://localhost:8081/myService 来表示 Web 服务，而不能使用 http://193.92.66.100:8081/myService。我们可以把这种自足的范例拿到客户所在的地方进行便捷的测试，也可以把它放在另外一个平台中测试（例如，从 Linux 平台转移到 Windows 平台），还可以将其发布在问答论坛上面（参见第 2 条），或是将其发给厂商，以寻求进一步的帮助。

此外，我们还需要在可以制作副本的执行环境下进行调试。如果不把正在调试的代码与运行该代码的系统固定起来，那么你可能会在一个根本就没有 bug 的地方去白费工夫。例如，我们要调试一款软件的安装程序。每次安装该软件时，操作系统的配置都会遭到修改，这是我们要在调试过程中竭力避免的事情。在这种情况下，可以考虑创建虚拟机镜像，该镜像中是一个干净的系统，可以供我们安装这款软件。每次安装失败之后，我们只需要从头开始使用那个干净的镜像就可以了。此外，也可以通过操作系统级别的虚拟化工具或容器工具（如 Docker）来达到类似的效果。如果能够装备一款系统配

置管理工具，那就更好了，如可以考虑 Ansible、CFEngine、Chef、Puppet 或 Salt 等。这些工具可以根据我们所发出的高级指令来可靠地创建特定的系统配置，从而简化生产、测试以及开发环境的兼容性维护工作，并且使我们能够像管控软件那样来管控它们的演化情况。

除了上述几点，我们还应该能够对发生故障的软件版本进行可靠的重制。为此，首先应该把软件置于 Git 这样的配置管理工具之下。然后，在构建的过程中，选取一个与构建所用的源代码版本有关的标识符，并把它放在软件的代码里面。下面这条 shell 命令能够打印一条变量初始化语句，这条语句会采用与最近一次的 Git 提交相对应的缩略哈希码来初始化 version 变量，你可以把它添加到源代码中。

```
git log -n 1 --format='const string version = "%h";'
```

例如，上面那条命令可能会输出：

```
const string version = "035cd45";
```

现在，我们可以给软件添加一种显示该字符串的方式，例如，可以通过命令行选项或 About（关于）对话框来展示 version 变量的值。有了这个字符串，我们就可以用下面这样的命令来获取与发生故障的软件版本相对应的源代码：

```
git checkout 035cd45
```

如果你在用旧代码构建软件时想要精确地重现当时的境况，那么别忘了把影响最终发行成果的所有元素全都纳入版本控制系统之下，如编译器、系统程序库、第三方程序库以及构建软件时所用的规范文件（如 Makefile 或 IDE 的项目配置文件）。最后，如果你需要把工具及运行时环境所带来的各种变化因素全都去掉，那么可以参考本书第 52 条的建议。

要点

- 如果能够准确重现程序中的问题，那么我们的调试过程就会得以简化。
- 创建一个简短且自足的范例，以便重现程序中的问题。
- 设法创建一套可以制作副本的执行环境。
- 采用版本控制系统给特定的软件版本打上标记，以便根据此标记来获取与之对应的代码。

第 11 条：修改完代码之后，要能够尽快看到结果

调试通常是一种循序渐进的过程。在每一轮中，我们都要花时间去构建并运行软件，而且要看着它发生故障，这些环节会占用很多时间，而且这些时间并没有用来解决软件中的问题。因此，我们要提前进行准备，设法缩短每一轮调试所花费的时间。

首先从软件的构建入手。我们应该能通过一条命令（如 make 或 mvn compile）或一个按键（如 F5）把发生故障的软件迅速构建出来。构建过程应该能够记录文件之间的依赖关系，使得我们在修改了某处代码之后只有少数几个文件需要重新编译。能够达到这种效果的构建工具包括 make、Ant 及 Maven。

高效地部署与运行软件也是相当重要的，然而各个项目的具体实现方式有很大的区别。我们可能要在远程主机上面部署文件、要重启应用程序服务器、要清除缓存，或要重新初始化某个数据库，为此，可以使用项目的构建系统，或编写一些脚本来自动执行该流程（参见第 12 条）。如果安装软件之前需要花很长时间来对分配文件进行构建，而且软件安装得也比较慢，那么可以考虑设置一种快捷的安装途径，使得我们只需把改动之后的那些文件复制到适当的位置即可。

最后，要确保软件能够尽快暴露出故障（参见第 55 条）。如果发生故障的代码可以进行单元测试，或是可以放在回归测试框架中进行测试，那就构建一个能够展示故障的测试用例（参见第 10 条），然后通过采用 IDE 或测试环境所提供的特性来运行这个测试用例。例如，如果代码是用 Maven 来管理的，那么可以通过下面这条命令来运行名为 TestFetch 的用例：

```
mvn -Dtest=TestFetch test
```

如果要调试的程序在处理某个特定的文件时会发生故障，那就构建一个能够引发该故障的最简文件。为了重现 GUI 应用程序中的问题，我们可以通过软件自动化应用程序来完成，例如，适用于网页浏览器的 Selenium、适用于 Windows 系统的 AutoHotkey、适用于 OS X 系统的 Automator，以及适用于 Linux 系统的 AutoKey。

要点

- 设法在修改代码之后尽快看到其结果，以提升调试的效率。
- 配置一套快速的自动化构建及部署流程。
- 测试软件时，要令其尽快地将故障暴露出来。

第 12 条：将复杂的测试场景自动化

我们可以用脚本对复杂的测试场景进行自动化。自动化的方式有很多种。如果是要对处理流程与文件进行编排，那么可以考虑 Unix shell 所提供的大量实用工具（参见第 22 条）。此外，通过能够获取 URL 的 curl 命令，以及能够解析 JSON 数据的 jq 命令，我们还可以用 shell 来测试 Web 服务。对于牵涉 API 访问及状态维护等事宜的复杂场景来说，我们可以求助于功能更为丰富的脚本语言，如 Python、Ruby 或 Perl，另外，还有很多系统会内置它们自己的脚本语言，如 Apache HTTP Server、Wireshark（一款网络包分析器）以及 VLC（一款媒体播放器）都支持 Lua 编程语言。

如果软件本身不支持脚本语言，但你能够对它进行修改，那么可以考虑将该软件与脚本语言结合起来，通过相关的 API 来把软件程序中的函数披露给脚本语言。现在看一个简单的例子（这是笔者特意构造出来的）。假设我们用 C 语言实现了一个数学函数库，并且需要对其进行测试，那么可以采用程序清单 2.1 这样的 C 语言代码来加载并运行名为 debug.lua 的 Lua 程序。这段 C 语言代码会把 sin、cos 及 tan 函数导出给 Lua 程序，以供其进行测试。

程序清单 2.1　把 C 语言函数导出给 Lua，以供其进行测试

```
#include <math.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

// Functions exposed to Lua
static int l_sin(lua_State *L) {
   double value_as_number = luaL_checknumber(L, 1);
   // Call the function, and return the result
   lua_pushnumber(L, sin(value_as_number));
   return 1; // Single result
}

static int l_cos(lua_State *L) {
   double value_as_number = luaL_checknumber(L, 1);
   lua_pushnumber(L, cos(value_as_number));
   return 1;
}

static int l_tan(lua_State *L) {
   double value_as_number = luaL_checknumber(L, 1);
   lua_pushnumber(L, tan(value_as_number));
   return 1;
}
```

```
int main() {
   // Setup Lua
   lua_State *L = luaL_newstate();
   luaL_openlibs(L);

   // Expose the functions to Lua
   lua_pushcfunction(L, l_sin);
   lua_setglobal(L, "lsin");
   lua_pushcfunction(L, l_cos);
   lua_setglobal(L, "lcos");
   lua_pushcfunction(L, l_tan);
   lua_setglobal(L, "ltan");

   // Load and run the debug file
   luaL_dofile(L, "debug.lua");
   puts("Done");
   return 0;
}
```

在 Debian Linux 系统中，可以通过 sudo apt-get install lua5.2-dev 命令来安装 Lua，并通过 cc myprog.c-llua5.2-lm 来编译上述 C 语言程序。（其他操作系统上面的安装方法，请参阅 Lua 的文档。）然后，笔者编写下面这个简短的 Lua 程序，并通过正弦函数的定义来验证数学函数库中的函数所达到的精确程度。

$$\tan\vartheta = \frac{\sin\vartheta}{\cos\vartheta}$$

```
epsilon = 1
errors = 0
while epsilon > 0 and errors < 2 do
   for theta = 0, 2 * math.pi, 0.1 do
      diff = lsin(theta) / lcos(theta) - ltan(theta)
      if (math.abs(diff) > epsilon) then
         print(epsilon, theta, diff)
         errors = errors + 1
      end
   end
   epsilon = epsilon / 10
end
```

运行 C 语言程序之后，该程序会加载 Lua 代码，并产生下面这样的输出：

```
1e-14 4.7   1.4210854715202e-14
1e-15 1.5   1.7763568394003e-15
1e-15 4.7   1.4210854715202e-14
```

在现实工作中，本例所举的 C 语言程序相当于你正在开发的大型应用程序，而受测的三角函数，则相当于应用程序里面有待检查的那些函数，你可以通过 Lua 程序为其

打造相应的测试用例。

要点

- 通过脚本语言来自动执行复杂的测试用例。

第 13 条：使自己尽可能多地观察到与调试有关的数据

我们在调试的过程中要处理大量数据，并且要把各式各样的数据关联起来，如源代码、日志文件中的条目、变量的值、栈的内容、程序的 I/O 以及测试的结果等。这些数据通常是由多个处理流程与计算主机所产生的，如果能够把它们全都适当地展示在调试者眼前，那么将会给调试工作带来很多的好处。首先，可以使我们发现数据之间的相互关系。例如，我们可以看到：当测试失败的时候，日志文件中会多出来一条记录。其次，它可以令人尽量保持专注，避免因为来回切换而使思路受到干扰。有时我们必须进入完全专注的状态（参见第 9 条），才能找到数据之间的关键联系，可是如果在进行单步调试的过程中总是要通过输入命令或切换窗口来查看某些变量的值，那么我们就无法保持这种专注的状态了。此外，留出足够的空间来显示较长的代码行，也能够帮助我们发现一些可能会错过的模式。你或许会按照很多代码风格指南所提倡的那样，把编辑器的窗口宽度设为 70 至 80 列，但是如果日志文件和栈跟踪信息中的某些行比较长，那么为了符合这个 80 列宽的限制，它们会分成很多行来显示，而这样是不利于进行阅读与分析的。因此，我们可以使编辑器在水平方向上尽量占满整个显示器，这样其中的某些模式就会自然地浮现出来，如图 2.1 所示。下面给出几条建议，希望能够帮助大家在调试时看到更多的数据。

第一条建议是：尽量扩大显示区域。很多人都使用两台或两台以上的高分辨率显示器来做开发。（用廉价的大屏幕电视机是不行的，因为它们显示出来的字符很模糊。）要想同时使用多台显示器，就必须配备功能强大的显卡接口。如果你用的是笔记本，那么可以外接一台显示器，把笔记本的屏幕延伸到那台显示器上（而不是使两者显示完全相同的画面），这样做可以增加你所能够看到的内容。无论是使用台式机还是笔记本，都应该把编辑器或终端机的窗口切换到全屏模式，这种显示效果在当前的全高清（full HD）显示器上面或许显得有点笨拙，然而对于某些调试任务来说却是必不可少的，因为它使得我们能够在横向和纵向上尽可能多地看到相关的数据。如果切换到全屏模式之后还是无法把数据显示到整个屏幕中，那么可以考虑把字体调小（并戴上眼镜），或是使

非卖品，仅供非商业用途或交流学习使用

用投影仪。

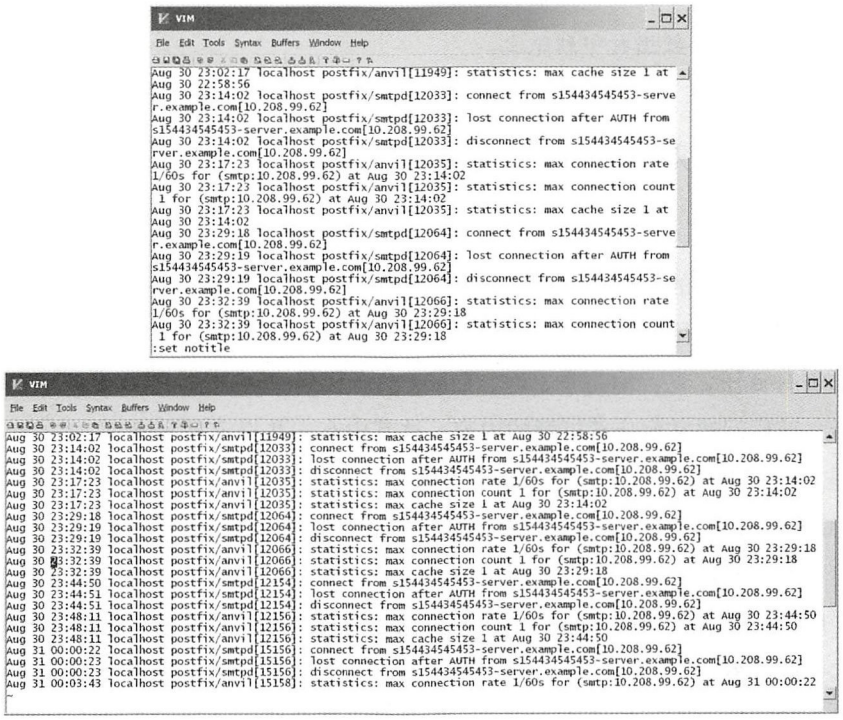


图 2.1　日志文件在宽度不同的编辑器中的显示效果。顶部是默认的宽度，底部是拉伸之后的宽度

把数据打印出来也是个相当有效的办法。激光打印机在 600dpi 的分辨率下，就可以把 6600 × 5100 像素的内容显示在与信纸大小相同的纸张上面。这样打出来的数据要比显示器更多，而且看起来也更为清爽。数据结构的定义以及代码清单等变化不太频繁的数据，可以打到纸上，以便给显示器留下更多的空间，用来展示那些在调试过程中频繁变动的数据。笔者最后要说的是，打印程序代码的最佳纸张，肯定是 15 英寸[⊖]的绿线条折叠纸（green bar fanfold paper），这种纸横向可以打 132 列，并且能够沿着纵向一直打印下去。如果你现在还有一台能够处理这种纸张的打印机，那就用它来打印代码吧。你要好好守着它。

要点

■ 如果能够同时看到比较多的数据，那我们就可以更加专注地进行调试，从而找

⊖ 1 英寸 = 0.0254 米。——编辑注

非卖品，仅供非商业用途或交流学习使用

到数据所体现出的模式以及数据之间的相互关系。

- 尽可能地将显示区域扩至最大。
- 把相对静态的数据打印到纸上。

第 14 条：考虑对软件进行更新

接下来，笔者要说几个很多人都意想不到的 bug 来源。并非所有错误都是由于你自己所写的代码而造成的，用来处理这些代码的编译器或解释器、你所使用的程序库、你所依赖的数据库和应用程序服务器，以及上述工具所在的操作系统，也有可能要对程序中的 bug 负有责任。笔者编写本书时，Linux 的源代码里面包含 2700 多条带有 XXX 字样的注释，该标记通常意味着可疑的内容，这其中有一些肯定是 bug。

于是，有些 bug 就可以通过更新软件来解决。如果你要发布的打包应用程序里面出现了隐晦的 bug，那么使用新版的编译器或程序库或许能够帮助你修复这个 bug。如果你要发布基于软件的服务，那么对中间件、数据库及操作系统进行更新也是很有好处的。至少应该尝试用最新的版本来构建、链接或运行程序代码，以尽量消除因第三方组件而引入 bug 的概率。然而，有时采用稍微保守一些的升级策略也是很有道理的，这就是说，尽管我们知道旧版软件不太完美，但是基于某些原因，我们还是决定继续使用这些软件。有很多中间件本身编写得就有错误，或是向后兼容性不太好，因此，有经验的用户通常会较为谨慎地进行升级，他们会升级到能够解决其问题，而且发布时间第二早的那个 bug 修复版本上面（如 6.4.3 版）。此外，新版软件也会引入新的 bug 与兼容性问题，因此在升级的时候至少要给自己留一条退路，也就是要拟定一套明智的后退计划，如果升级失败，或是升级到的版本无法解决你所面对的 bug，那么可以根据该方案退回原有的版本。我们可以在沙盒中更新第三方的代码，例如，可以对虚拟机进行复制，在复制出来的镜像里面进行更新，如果更新后的效果不好，那么直接把这份镜像丢掉就行了，这是一种可靠且较为简便的办法。尽管更新软件对解决 bug 确实有一定的帮助，但无论如何都不要对此抱有过高的期望。

除非你能够证明外部代码确实有错，否则最好是先假设它们正确无误。有些 bug 看上去好像是由第三方代码所引发的，然而在大多数情况下，其实是由于你自己的问题而造成的。30 年的调试历程中，笔者在自己的代码里面修复了数以千计的 bug，与之相对，由于某款流行的商业编译器生成了错误的代码而导致的 bug 只出现了一次，由于程序库

非卖品，仅供非商业用途或交流学习使用

而导致的 bug 只出现了几次，由于操作系统的功能不稳定而导致的 bug 只出现了一次，由于描述系统调用的文档有错而导致的 bug 只出现了几次，由于工具和其他系统软件而导致的 bug 也只出现了几十次。因此，对软件进行更新的最大意义，就在于给我们提供一个新的起点，令我们可以下定决心来好好清理自己的代码。

要点

- 在更新之后的环境里面重新尝试你所编写的代码，看看这次会不会出错。
- 不要对更新软件所带来的效果抱有过高的期望。
- 要考虑因第三方组件而引发 bug 的可能性。

第 15 条：查看第三方组件的源代码，以了解其用法

我们所要调试的代码之所以会出 bug，通常并不是由于它使用的第三方程序库或应用程序本身有问题（参见第 14 条），而是因为它使用这些第三方组件时所采取的方式有误。

这种情况并不令人惊讶，由于这些软件本身是作为黑盒来与你所写的代码进行集成的，因此，你不太可能在它们之间相互协调。对于这类问题来说，有一个很有用的办法，就是去查看第三方程序库、中间件甚至是底层软件的源代码。

首先，如果想查明某个 API 为什么没有像你所期望的那样运作，或是想查明某条奇怪的错误消息是从哪里发出来的，那么可以浏览第三方组件的源代码，并搜寻其中你所感兴趣的那一部分，以寻求答案。如果想对某个与程序库有关的功能进行调试，那么可以在第三方代码中寻找相关函数或方法的定义，然后由此继续向下查看。在查看第三方程序库的代码时，我们可能并不是要在其中寻找 bug，而是想更好地理解它的工作原理以及它与你所写的代码之间的配合方式。如果想知道某条错误消息是从哪里来的，那么可以在整个代码库里面搜寻这条消息，并检查有哪些代码会发出这条消息（参见第 23 条和第 4 条）。为了能够迅速地查询函数或方法，我们可以用 ctags 或 etags 程序为代码编制索引（大多数编辑器都支持这两个程序所输出的索引），也可以采用集成开发环境（IDE）来查看代码。与 ctags 相比，IDE 能够更好地处理复杂的语言特性，如重载、覆盖以及模板等，而 ctags 的优势则在于支持的语言数量比较多，5.8 版本支持 41 种语言。在存放源代码的目录里面运行下面这条命令，可以给其中的所有文件创建索引：

非卖品，仅供非商业用途或交流学习使用

```
ctags -R .
```

如果你所使用的第三方组件是开源的，那么也可以通过 Black Duck Open Hub Code Search 这样的托管服务来进行搜索。

还有一种更为强大的调试技术，需要我们先构建带有调试信息的第三方组件（参见第 28 条），然后把自己的代码与这个调试版的第三方程序库相链接。这样，我们就可以像调试自己的代码那样，轻松地在第三方程序库的代码里面进行单步调试了，而且我们还可以使用符号调试器（symbolic debugger，参见第 4 章）来检视其中的变量。请注意，某些厂商（如 Microsoft）在发行其代码的时候，会把供调试所用的版本或带有符号的版本也随着普通版本一起发布出来，这样我们就不用自己去构建包含调试信息的程序库了。

如果 bug 确实是由第三方代码而非自己的代码所引起的，那么在能够访问其源代码的前提下，你可以对此进行修改。请注意，只有在极其特殊的情况下，才需要使用这种办法，也就是说，只有在既找不到合理的权宜方案又联系不到厂商来对此进行修复的时候，才可以考虑修改第三方程序库的源代码。一旦你这么做，就必须在整个应用程序的生命期内，对该程序库的所有后续版本都做出相应的修改。此外，你还需要保证这样的改动不会违背相关的法律条款。例如，某些厂商在发布代码的时候，会在其协议中要求“只能看，不能改”（look, don't touch）。对于开源软件来说，比较合理的一种做法，是将修改后的代码提交给负责该代码的项目，而且从道理上来讲，你也确实应该这么做。如果项目托管在 GitHub 上，那么只需要发起 pull request 就可以了。

要想运用上面提到的技术，我们必须先获取到第三方组件的源代码。如果第三方程序库或应用程序是开源的，那么很简单，只需要点一下按钮，就可以把源代码下载下来了。对于开源的操作系统发行版，还可以把源代码作为软件包来下载，例如，在 Debian Linux 系统中，我们可以通过以下命令安装 C 语言程序库的源代码：

```
sudo apt-get install glibc-source
```

此外，很多软件开发平台也会将其源代码中较为重要的那一部分，安装到你的系统上面。例如，Microsoft 的 Visual Studio 会把 C 语言运行时库的源代码放在 VC\crt\src 里面，而 Java Development Kit（JDK）则会把源代码放在名为 src.zip 的压缩文件中。对于其他一些第三方软件来说，你可以在订购软件时支付额外的费用以获取其源码。只要售价不高，我们就应该考虑这么做，若是等以后再去买，则要花很多时间来安排款项，而

非卖品，仅供非商业用途或交流学习使用

且还要下订单并执行所需的合约。等到那个时候，软件厂商或许已经不再支持你所使用的版本了，甚至有可能已经停业了。因此，我们应该考虑提前获取这些专有软件的源代码，以免出现此类问题。

要点

- 如果你依赖某个第三方组件，那么就应该获取其源代码。
- 通过查看第三方组件的源代码探寻与第三方 API 及一些奇怪的错误消息有关的问题。
- 要和第三方程序库的 debug 版本相链接。
- 只有当其他办法都不可行的时候，才需要对第三方的源代码进行修改。

第 16 条：使用专门的监测及测试设备

调试嵌入式系统及系统软件的时候，我们可能要对从硬件到应用程序的整个计算栈进行分析。调试工作一旦深入硬件层面，我们就需要关注电流的微小变化以及磁矩的对齐情况等细节。在大多数情况下，可以通过强大的 IDE 以及一些追踪软件与日志记录软件来探查这些问题，然而有的时候，就连这些工具也帮不上忙。这通常发生在软件与硬件有所接触的场合，也就是说，虽然你认为你所写的软件能够像预期的那样运作，但是硬件却有着它自己的处理方式。例如，你把正确的数据写入磁盘，再将其读取出来，却发现这些数据似乎已经损坏了。在调试这些接近于硬件层面的问题时，某些奇特的设备有可能给你提供很大的帮助。

有一种通用的工具或许可以派上用场，这就是逻辑分析仪，它能够以每秒数百万次的采样率来捕获、存储并分析数字信号。这些设备原来卖得比汽车还贵，但是现在只要 100 美元左右就可以买到一台基于 USB 的逻辑分析仪了。它不仅能够监测主板上面的所有数字信号，而且还能对组件之间进行通信时所采用的一些高层通信协议进行监测。某家制造商（saleae）所出售的产品支持很多种通信协议，其产品信息中罗列了大量的首字母缩略词：“SPI、I2C、serial、1-Wire、CAN、UNI/O、I2S/PCM、MP Mode、Manchester、Modbus、DMX-512、Parallel、JTAG、LIN、Atmel SWI、MDIO、SWD、LCD HD44780、BiSS C、HDLC、HDMI CEC、PS/2、USB 1.1、Midi”。

如果你专门研究某个特定的技术，那么可以考虑购买协议分析仪或总线分析仪等专

非卖品，仅供非商业用途或交流学习使用

用的设备。例如，车辆以及其他机器中的微控制器，一般是通过 CAN（Controller Area Network，控制器局域网）总线来通信的。有很多公司都在出售那种可以独立运作的分析仪器，把这些仪器插入总线之后，它们就能够对流经总线的数据进行过滤、显示及记录了。对于其他一些使用范围较广或者更加专业化的物理连接方式及协议（如 Ethernet、USB、Fibre Channel、SAS、SATA、RapidIO、iSCSI、sFPDP 及 OBSAI）来说，也有类似的产品可供购买。与基于软件的解决方案相比，这些设备应该能够在其所宣称的线路速率之下工作，而且它们通常可以同时监测多个数据通道，并允许使用者根据位于数据包深处的位模式（bit pattern）来定义触发器与过滤器。

如果没有专门的硬件来进行调试，那么你应该临时打造一个符合自身需求的设备，以帮助你探查那些难以重现的问题。几年前，笔者遇到过这样一个问题：以网页表单的形式所实现的投件箱，可能会丢失数据。这个问题的难处在于：尽管它在数天之间影响了很多用户并招致其抱怨（该程序有成千上万的用户），但是在我们这里，它的出现概率却很低，几乎无法重现。对受影响的用户所在的地方进行分析之后，我们发现这些用户大多位于偏远地区。于是，我们认为问题可能与网络连接的质量有关。我拿了一个 USB 无线猫，把它用锡纸裹起来，以模拟信号强度较弱的情况，然后通过这个无线猫来连接应用程序的 Web 界面，这次果然就出现了那个问题，由于该问题已经能够很方便地重现出来（参见第 10 条），因此只过了几个小时我们就把它解决了。

如果你想调试的代码，是以嵌入式软件的形式来运行的，并且运行该软件的设备又缺乏适当的 I/O 机制，那么可以考虑采用下面这些技巧来与正在调试的软件进行通信。

- 如果设备有状态指示灯或能够发出响声，那么可以用特定的闪烁方式或发声方式来表达软件当前的状况。例如，可以用响一声来表示软件进入了某个特定的例程，用响两声来表示软件离开了该例程。你也可以用摩尔斯电码发送更为复杂的消息。
- 把输出到日志文件中的信息先保存到非易失性的存储器（non-volatile storage）中（甚至可以保存到外接的 U 盘中），然后将其导入自己的计算机，并对其进行分析。
- 实现一个简单的串行数据编码器，将数据写入某个尚未使用的 I/O 针脚，然后对信号进行电平转换，将其转为 RS-232 标准，并通过串口转 USB 口的适配器及终端应用程序来在计算机上读取这些数据。
- 如果设备有网络连接，那么显然可以通过该连接来通信。如果设备缺乏对网

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

络日志（参见第 41 条）或远程 shell 访问提供支持的软件，那么可以考虑通过 HTTP 甚至 DNS 请求来与外界通信。

要想对网络数据包（network packet）进行监测，就应该先把网络硬件设置好，令其可以接受软件形式的网络包分析器，如开源的 Wireshark 等。我在笔记本上运行的 Wireshark 版本支持 1514 种（网络与 USB）协议及数据包类型。如果能把要调试的程序与 Wireshark 放在同一台计算机中运行，那就可以轻而易举地监测网络数据包了。我们只需要把 Wireshark 打开，指定自己想要捕获的数据包，然后等着看详细信息就行了。

对两台计算机之间的网络通信（例如，应用程序服务器与数据库服务器或负载均衡器之间的通信）进行监测，或许要显得困难一些。由于这两台计算机可能都要把网线连接到交换机（switch）上面，因此，两个相关端口之间的通信就必须通过交换机才能进行。这个问题可以用很多种办法来解决。

如果你所在的组织使用的是管理型交换机（managed switch，这种交换机比较贵），那么可以对其进行设置，使它把某个端口镜像到另外一个端口上面。也就是说，可以把服务器流量所经过的端口镜像到我们自己的计算机所使用的端口上面，并通过计算机中运行着的 Wireshark 软件对流经的数据进行捕获及分析。

如果没有这种管理型交换机，那就试着找一台集线器（hub），这种设备比前者便宜很多，它会把接收到的以太网数据广播给所有的端口。由于现在已经不再生产集线器了，因此它通常要比廉价的交换机贵一些。把待监测的计算机和运行着 Wireshark 软件的计算机连到同一个集线器上面，就可以进行监测。

使用 tcpdump 这样的命令行工具，也可以对远程主机进行监测。我们需要登录到待监测的远程主机上，并运行 tcpdump 命令，以查看自己所关注的网络数据包。（该命令需要管理员权限才能执行。）如果还想通过 Wireshark 的 GUI 对此进行深入分析，那么可以先用带有 -w 选项的 tcpdump 命令将原始数据包写入文件中，然后再通过 Wireshark 详细地分析它。这种工作方式很适合用来监测基于云端的主机，因为你不太容易去调整其网络配置。

最后一种办法是配置一台计算机，并用该计算机对需要监控的计算机与网络中的其余计算机进行桥接。我们给这台计算机配置两个网络接口（例如，用其中一个接口来表示插有网线的网络端口，用另一个来表示基于 USB 的端口），并将这两个端口桥接起来。在 Linux 系统中可以通过 brctl 命令来实现，在 FreeBSD 系统中可以通过 if_bridge 驱动来配置。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

此外，我们还可以考虑对设备进行配置，以模拟各种不同的网络场景，例如，可以模拟数据包从世界各地的主机中传过来的情况，或是模拟流量整形（traffic shaping）、带宽限制以及防火墙配置等。此时所需的软件是运行在 Linux 系统中的 iptables。

要点

- 逻辑分析器、总线分析器或协议分析器可以帮你锁定接近于硬件层面的问题。
- 可以通过自制的设备来探查与硬件有关的问题。
- 可以通过将 Wireshark 与以太网集线器相结合、使用管理型交换机或进行命令行捕获等办法来监测网络数据包。

第 17 条：使故障更加突出

使故障显得更加突出一些，可以提升调试工作的效率。为了突出软件的故障，我们可以对软件本身，或是对其输入数据所在的环境进行修改。无论怎样修改，都必须在版本控制系统中进行，而且要在单独的分支上面修改，这样，以后可以轻易地恢复到原来的版本，并且不会在产品代码中引入错误。

有时无论怎么调试，软件都无法像我们期望的那样运作。例如，尽管某一组复杂的条件显然可以得到满足，但数据库里面就是没有出现我们想看到的那条记录。在这种情况下，有一个比较好的办法，是大幅度地改变软件的执行路径，并看其运行结果与你所想的是否相符。如果不符，那说明你思考方向可能有误。

下面这段代码节选自 Apache HTTP 服务器，它用来处理签名证书的时间戳（signed certificate timestamp，SCT）。服务器有时可能无法应对比当前日期还大的 SCT。

```
for (i = 0; i < arr->nelts; i++) {
   cur_sct_file = elts[i];
   rv = ctutil_read_file(p, s, cur_sct_file, MAX_SCTS_SIZE,
             &scts, &scts_size_wide);
   rv = sct_parse(cur_sct_file,
             s, (const unsigned char *)scts, scts_size, NULL,
             &fields);
   if (fields.time > apr_time_now()) {
      sct_release(&fields);
      continue;
   }
   sct_release(&fields);
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    rv = ctutil_file_write_uint16(s, tmpfile,
              (apr_uint16_t)scts_size);
    if (rv != APR_SUCCESS)
       break;
    scts_written++;
}
```

要想对这种情况进行调试，我们可以临时修改判断条件，使其总是为真（true）。

```
if (fields.time > apr_time_now() || 1) {
```

这样修改之后，我们就可以判断出问题到底是出在修改之前的 Boolean 条件上面，还是出在测试数据或是用来处理“SCT 大于当前日期”的那部分逻辑上面。

还有一些类似的技巧可以考虑，例如，在方法的开头添加 return true 或 return false 语句，或是用 if(0) 把某段代码包裹起来，使得程序跳过这段代码（参见第 46 条）。

有时我们可能要调试一种或隐或现的效果。在这种情况下，可以临时修改代码，使得这种效果展示得更为明显一些。例如，在制作游戏时，我们想令某个角色在经历了某种事件之后的一分钟之内，具备能力提升的效果。如果在发生了相关事件之后，我们无法看出该角色的能力是否得到了提升，那么可以大幅度地增加能力的提升值，这样就可以更好地进行观察了。例如，在 CAD 程序里面计算地震对建筑物的影响时，我们想对这种计算逻辑进行调试，那么可以把显示出来的结构位移放大 1000 倍，以便更好地看出建筑物的移动幅度及移动方向。

如果软件的故障依赖于外部因素，那么可以修改软件的执行环境，令其更为迅速或更为频繁地出现错误（参见第 55 条），从而凸现该故障。如果软件要处理的是 Web 请求，那么可以运用 Apache JMeter 这样的负载测试（load test）或压力测试（stress test）工具，把软件推进到一种行为有可能开始出现异常的地步。如果软件是通过多线程来实现并发的，那么可以增加线程的数量，使其超过计算机的 CPU 核心数量所能担负的值，以促使程序出现死锁（deadlock）及竞争条件（race condition）等问题。此外，也可以同时运行其他一些耗费内存、CPU、网络或磁盘资源的进程，迫使你所开发的这款软件去和那些进程争夺稀缺资源。其中特别有效的一种做法，是令软件把数据写入容量很小的 U 盘中，看看它在没有磁盘空间的情况下会表现出什么行为。

最后，还有一种办法也可以帮助你查出较为罕见的数据验证或数据损坏问题，这就是模糊测试（fuzzing）。要想采用这种办法来进行测试，我们可以把随机生成的一些值输入给程序，也可以随机地扰乱程序的输入值，然后看看程序会表现出何种行为。我们

非卖品，仅供非商业用途或交流学习使用

的目标，是要找到一种能够令程序偶尔发生故障的数据模式，并采用系统化的方式来提升这种数据模式的出现概率。做到了这一点之后，我们就可以拿这些有问题的数据来调试程序了。如果应用程序在客户的计算机上运行并处理其生产数据时总是发生故障，而在你自己的开发计算机上运行时却不会发生故障，那么这个技巧或许能够帮你找到原因。模糊测试可以通过 zzuf 这样的工具用来执行。

要点

- 迫使软件去执行那些可疑的路径。
- 提升某些效果的幅度，令其变得更加突出，以便于我们进行研究。
- 对软件加压，迫使它走出能够从容应对负载的那种舒适状态。
- 在版本管理系统中临时创建一个分支，并把所有的修改都放在这个分支上面来做。

第 18 条：从自己的桌面计算机上调试那些不太好用的系统

Jenny 和 Mike 在谈论各自的调试经历。Jenny 说：“我不喜欢在客户的计算机上面工作，要用的工具都没装，浏览器里也没有我收藏过的书签。这真是太麻烦了。我访问不了自己的文件，计算机上的按键绑定和快捷键，设置得也都不对。”Mike 惊讶地看着她说：“快捷键？你还有快捷键可用，这都算不错的了。我调试的那台计算机，连键盘都没有！”

如果你在工作时无法使用自己配置好的这台计算机，那么工作效率确实会大幅降低。除了 Jenny 抱怨的那些事情，还会出现其他一些琐碎的问题，例如，连接因特网或内部网络时受到限制、计算机的配置方式不符合习惯（其中既包括屏幕与座椅等设施的位置，也包括鼠标与键盘等设备的选用）、必须经过长途跋涉到达气候湿热（或寒冷）的偏远地点，以及计算机性能过低等。这些问题目前都特别常见，而且考虑到业界对移动设备及物联网的热情，它们以后还会更加普遍。在很多情况下，你都必须离开自己所习惯的这台高端计算机，例如，要调试手机 App，要调试带有嵌入式软件的设备，要解决只会在客户的计算机上面出现的问题，或是要处理数据中心里面的紧急状况等。在这些情况下，其实你依然有办法可以继续使用自己所熟悉的键盘，只不过你要提前做好准备。

非卖品，仅供非商业用途或交流学习使用

对于手机 App 和某些嵌入式设备来说，我们可以在计算机上面通过设备模拟器（device emulator）来调试有问题的应用程序。然而从调试的角度来看，这种方式除了可以提供一些先进的日志记录功能，并不会给我们带来太大的帮助。有了模拟器之后，确实不需要再触摸手机屏幕上的键盘了，可是我们无法在模拟器里面运行符号调试器。比较好的一点是，我们可以同时在计算机里面打开模拟器与包含源代码的编辑器，这样就能够在修改完代码之后迅速看到结果，而不用再将其部署到实际的设备上面了。

还有一种更为强大的办法，是创建 software shim（软件楔子），以便将应用程序中较为关键的那些部分，放在自己的计算机里面来运行，为此，我们经常会用到单元测试及 mock 对象（参见第 42 条）。这种办法虽然不能够使用图形界面，但是却可以把一些棘手的算法轻松地包含进来，这些算法需要进行大量的调试才能够处理好。我们可以在应用程序的算法里面设置挂钩，将其关联到某种简单的（如基于文件的）输入 / 输出上面，这样就可以在自己的计算机上面直接编译并运行源代码了，而且稍后也可以借助功能强大的调试器来对其中较为复杂的那一部分进行单步调试。

例如，我们要做一个手机 App，把社交网络上面的朋友照片，导入手机的联系人中。这个程序有一个比较困难的地方，就是要和社交网站通信，并与手机中的联系人配对。于是，我们可以写一个命令行工具来充当 shim，该工具的参数是某位联系人的名字，它会采用 Facebook/LinkedIn/Twitter 等网站的 API 来获取相关的信息，并在其中寻找与该联系人相匹配的资料。等我们把这部分逻辑调试好之后，就可以把它作为一个类，集成到手机 App 里面了。请注意，这部分代码始终都应该能够作为独立的命令来运行（例如，可以通过在类中编写 main 方法来做到这一点），因为以后如果出了问题，我们可以直接编译并运行这个命令。

要处理好与远程访问有关的事宜，以便能够远程地解决客户计算机上面的问题。由于远程访问通常需要管理员权限和一些专业的技术知识，因此要提前做好准备。尽管有很多操作系统都提供了远程访问其桌面的功能，但是技术支持人员一般还是愿意使用 TeamViewer 等专门的应用程序。此外，还可以考虑在客户的计算机上面安装一些可以简化调试工作的数据和工具。例如，安装一个查看器来检视应用程序中的二进制文件，或者安装一个运行跟踪器。如果一定要在第三方计算机上安装一款调试工具，那么笔者会选择 Unix 的 strace 或 truss 命令。顺便说一句，像我们这些做 IT 的人，经常会有朋友和家人请我们帮忙去解决计算机问题，在这种情况下，远程访问也可以简化问题的解决过程。

非卖品，仅供非商业用途或交流学习使用

当前有很多后端运算，都是通过云主机提供商所搭建的机制来完成的，它们会提供漂亮的 Web 界面，使得开发者可以在其中进行调试并访问控制台。如果你要调试的服务器没有放在这样的云主机提供商那里，而是位于阴冷、嘈杂且不便访问的数据中心，那么你就得提前做好打算了。由于有些问题发生在与服务器建立网络连接之前，因此我们通常要使用服务器自身的屏幕及键盘，才能够解决这些问题。为了免去这种麻烦，可以考虑采购 KVM over IP 设备，它能够通过 IP 网络来远程访问计算机的键盘（keyboard）、显示器（video）及鼠标（mouse）。如果能够安装、配置并测试好这样一套设备，那你就可以在自己的桌面计算机上，对远程服务器在启动过程中所出现的问题进行调试。

要点

- 把设备模拟器配置好，以便通过计算机屏幕和键盘来调试移动 app。
- 搭建 shim 机制，以便使用自己计算机中的工具来调试嵌入式代码。
- 为远程访问做好准备，以便能够远程调试客户的计算机。
- 配置 KVM over IP 设备，以便调试远程服务器上面的问题。

第 19 条：使调试任务自动化

我们或许会找到很多个与程序错误有关的因素，但是却没有办法轻易推断出究竟哪一个因素才是致使程序出错的真正原因。为了把这个原因找出来，我们可以编写一小段例程或脚本，把有可能使程序出错的所有情况全都搜索一遍。如果待搜索的情况比较多，不便于手工进行搜索，但是却能够通过循环来进行遍历，那么就可以考虑对其加以自动化。例如，如果想遍历的是 500 个字符，那么可以通过自动化的脚本来实现，然而如果要把用户可能会输入的所有字符串全都尝试一遍，那么采用自动化脚本就不太合适了。

下面举一个例子。假设在某次升级之后，我们发现，计算机执行 which 命令的速度比原来慢了，然而把那一串比较长的搜索路径（Windows 和 Unix 系统用 PATH 变量来代表该路径）改成 /usr/bin 之后，which 命令的速度又快了起来。于是问题就来了：在该路径的 26 个元素中，到底是哪一个元素拖慢了 which 命令的速度？下面这个 Unix shell 脚本，可以显示出 which 命令在单独采用路径中的每一个组成部分时，所分别消耗的时

非卖品，仅供非商业用途或交流学习使用

间（我们通过 Cygwin，在 Windows 系统里面运行这个脚本）。

```
# Obtain path
echo $PATH |
# Split the :-separated path into separate lines
sed 's/:/\n/g' |
# For each line (path element)
while read path ; do
  # Display elapsed time for searching through it
  PATH=$path:/usr/bin time -f "%e $path" which ls >/dev/null
done
```

脚本所输出的其中一部分内容如下：

```
0.01 /usr/local/bin
0.01 /cygdrive/c/ProgramData/Oracle/Java/javapath
0.01 /cygdrive/c/Python33
4.55 /
0.02 /cygdrive/c/usr/local/bin
0.01 /usr/bin
0.01 /cygdrive/c/usr/bin
0.01 /cygdrive/c/Windows/system32
0.01 /cygdrive/c/Windows
0.01 .
```

由此我们可以发现，问题出在只含有一个斜线符号（/）的元素上面，这个斜线不小心跑到 which 命令的搜索路径里面来了。然后，我们对 which 命令的执行方式进行追踪（参见第 58 条），于是就找到了问题的根源：由于 which 命令会在路径中的每个元素后面补充一个斜线，因此，本来只包含一个斜线的那个元素，在搜索的时候就相当于变成了双斜线（//），而 Windows 系统如果遇到了这种以双斜线开头的路径，则会触发查找网络驱动器的流程。

如果你正在调试的这款软件，很难通过脚本来执行穷举式的搜索，那么可以考虑在程序中嵌入一个小的例程，以实现相同的目标。该例程通过某种算法（例如，对某些值进行迭代），把有待测试的各种情况全都生成出来。此外，也可以考虑使程序从某个外部文件中，把这些有待测试的情况读取进来，这样就可以用更为复杂的脚本或执行日志中的某些数据来生成那个外部文件了。

最后，我们还可以用一些工具来评估代码，以检测其中的 API 违例、内存缓冲区溢出以及竞争条件等问题（参见第 59 条和第 62 条）。用了这些工具之后，原本只需几秒钟就能运行完的测试分析任务，现在可能需要几十分钟，然而这是值得等待的，因为它们毕竟能够帮你省下大量的时间。

要点

- 把寻找程序故障的过程自动化，使得计算机多费一些功夫去搜寻，从而节省你自己的宝贵时间。

第 20 条：开始调试之前与调试完毕之后都要把程序清理干净

如果你要调试的软件有 10 个地方可能出错，那么这些错误就会有上千种（2 的 10 次方）表现形式，如果有 20 个地方可能出错，那么表现形式就会高达一百多万种（2 的 20 次方）。因此，在调试的时候，应该优先关注当前区域中最容易解决的那些问题。例如：

- 能够借助工具而找到的问题（参见第 51 条）。
- 程序在运行时所产生的警告（例如，可恢复的断言失败）。
- 读起来比较费解，而且与你要调试的 bug 有所关联的那些代码（参见第 48 条）。
- 标有 XXX、FIXME 及 TODO 字样，或是注释中写有“它应该会……”（should）、“我想它会……”（think）以及“它必定会……”（must）等托词的可疑代码。
- 其他一些已知但是却容易忽略的小 bug。

如果连一个相对无错的环境都没有搭建好，就匆忙地去调试那些棘手的问题，那么很可能要遭受惨痛的失败。

有人也可以举出理由来反对这种做法。首先，这种做法与“东西没坏就别修”（If it ain't broke, don't fix it）的理念不相符。其次，由于我们可能只是用比较新的写法，修改了系统代码中的某一部分，因此，整个代码的风格或许会显得有些失调。面对这些理由，你需要做出自己的判断。如果你认为清理代码肯定能帮助自己调试某个难以捕获的 bug，那就应该承担这种风险；反之，若是代码本身就很脆弱，而且你也明明知道自己能够直接找到这个 bug，那就没有必要去冒险清理代码了。

找到并修复程序错误之后，不要急着去做其他事情，因为你还有两项任务没有完成。第一，要在代码中寻找类似的错误，并将其修复（参见第 21 条）。第二，要把寻找问题时所做的那些修改整理好（参见第 40 条）。有时我们为了凸现错误效果，可能会临时改动一些代码，这些改动现在应该予以还原。如果你是在版本控制系统里面，单独用一个分支来进行调试的（参见第 26 条），那么还原起来应该很方便。此外，我们在修改过程中所添加的一些代码，以后有可能还要用到，因此要把这些代码清理干净并提交上

去，例如，断言、log 语句以及新的调试命令等。

要点

- 在开始调试重大的 bug 之前，先要确保代码能够达到一定的整洁程度。
- 调试完毕之后，要把调试过程中对代码所做的临时改动还原回去，并且要把那些有用的代码提交到代码库。

第 21 条：把属于同一个类型的所有问题全都修复好

在某一个地方所发生的问题，也有可能出现在其他地方。之所以会这样，可能是因为开发者在这些地方都采用了相同的思路来编程，也有可能是因为使用了某个很容易遭到误用的 API，或把错误的代码从一个地方复制到了其他很多地方。在许多较为成熟的开发环境或对安全要求很高的工作场合中，开发者并不会在修复了某一个问题之后就止步于此，而是会把属于同一类型的所有问题全都解决好，以防将来再出现类似的错误。

例如，你发现下面这条语句的除数可能会是 0，并且已经将该问题解决了：

```
double a = getWeight(subNode) / totalWeight;
```

那么，接下来你还应该搜索整个代码，看看有没有其他地方也把 totalWeight 用作除数。你可以通过 IDE 或 Unix 的 grep 命令（参见第 22 条）来搜索：

```
# Find divisions by totalWeight, ignoring spaces after
# the / operator
grep -r '/ *totalWeight' .
```

做完了这一步之后，还应该考虑代码中有没有其他地方也会出现类似的除法问题。我们需要寻找并修复这些可能出错的地方。借助 Unix 的管道机制，可以轻松地实现这种搜索。下面这段命令，能够在四百万行 C 语言代码里面，把可能出现除法问题的地方找出来。

```
# Find divisions, assuming spaces around the / operator
grep -r ' / ' . |
# Eliminate those involving sizeof
grep -v '/ sizeof' |
# Color divisors for easy inspection and
# eliminate divisions involving numerical or symbolic constants
grep --color=always ' / [^0-9A-Z][^,;)]*' |
```

```
# Remove duplicates
sort -u
```

经过几次过滤之后，可疑的代码从 5731 行降为 5045 行，又降为 2032 行，最后只剩下 1923 行。这个代码量，使得我们可以在合理的时间范围之内，将其中的代码审视一遍。尽管过滤得并不是特别严谨（例如，sizeof 有可能返回 0、符号常量的值有可能是 0），但毕竟可以使我们知道其中有哪些代码可能出现问题。这要比那种以工作量过大为借口而根本不去检查除法代码的做法好很多。

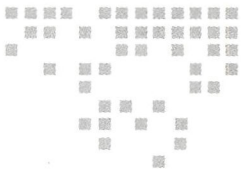
最后我们还要考虑的是：怎样在将来的工作中避免类似的错误。这可能需要我们对代码或者软件开发流程做一些调整。例如，如果程序总是由于误用某个 API 函数而出错，那么可以提供一个更为安全的版本，并且把原来的版本屏蔽掉。例如，在项目的全局 include 文件里面，可以写上这么一行代码：

```
#define gets(x) USE_FGETS_RATHER_THAN_GETS(x)
```

根据上述定义，如果有人试图在程序里面使用 gets 函数（该函数很容易发生缓冲区溢出的问题），那么代码就无法编译或链接。如果程序是因为对类型错误的值进行处理而出错的，那么可以考虑进行更为严格的类型检查。此外，还可以通过运用静态分析技术或更为严谨的配置方案来找到许多程序错误（参见第 51 条）。

要点

- 修复了某一个错误之后，我们还需要寻找并解决其他相似的错误，并设法保证将来不会再出现此类错误。



第 3 章 Chapter 3

通用的工具与技术

某些专用的调试工具用起来很方便，而且效率很高，但是通用的调试工具依然有其优势，因为它们可以迅速解决与各种语言及平台有关的许多开发和操作问题。本章所描述的工具源自 Unix，不过当前已经可以在 GNU/Linux、Windows 及 OS X 等大多数操作系统上面运行了。这些工具灵活、高效，而且应用范围较广，因此大家应该花一些工夫去掌握它们。“ The Art of Command Line”项目可以帮助你了解这些工具的用法，该项目是由 Joshua Levy 创建，并由多人协作编辑的。由于笔者假设你已经知道 Unix 命令行及正则表达式的基础知识，因此我们直接开始讲解怎样用这些工具与方法进行调试。

第 22 条：用 Unix 命令行工具对调试数据进行分析

调试程序的时候，可能会碰到原来从未遇见的问题。编写软件所使用的 IDE，或许缺乏足够强大的工具来供你详细探查这些问题。这正是 Unix 命令行工具可以派上用场的地方。这些通用的工具可以通过管道进行丰富的组合，从而令你能够轻松地对文本数据进行分析。

在开发者所经手的各种调试数据中，最有用且最常见的形式，就是基于文本行的数据流。这种数据流可以用来表示调试时所遇到的各类数据，例如，程序的源代码、程序的日志、版本控制系统的历史记录、文件清单、符号表、压缩文档的内容、错误消息、

测试结果以及与性能分析有关的数据等。对于经常需要执行的很多种任务来说，你可能更愿意采用强大而万能的脚本语言来处理数据，如 Perl、Python、Ruby 或 Windows PowerShell 等。如果脚本语言的界面能够较好地获取待调试的数据，而且你也能够较为流畅地用该语言进行交互式的开发，那么这种办法就比较合适，否则，你可能就必须编写一个小型的独立程序，并将其保存成文件了。这样做或许比较枯燥，促使你又改回原来那种手动调试的办法。这会令你无法在调试过程中抓住一些关键的内容。

一般来说，更为有效的做法应该是把各种 Unix 工具连成一条简洁的管道，从而可以在命令提示符界面中运行。由于当前的 shell 提供了较好的命令行编辑能力，因此我们可以逐步打造命令，直至其符合需求。

本条目将会概述怎样用 Unix 命令来处理调试数据。如果你不太熟悉命令行的基本知识与正则表达式的用法，那么请参考网上的教程。此外，你也可以把该命令的名称当作参数，传给 man 命令，以查看其各种选项的用法。

有一些操作系统可以轻而易举地访问 Unix 命令行界面，另外一些系统配置起来也不是特别困难。在 Unix 与 OS X 系统中，只需要打开终端窗口，就可以执行 Unix 命令了。而在 Windows 系统中，最好的做法则是安装 Cygwin，这是一款移植到 Windows 平台上面的软件包管理器，其中包含很多 Unix 工具，它的功能比较强大，而且运行得也较为流畅。接下来会提到几款没有默认安装在 OS X 系统中的工具，Homebrew 软件包管理器可以简化这些工具的安装过程。

我们会用 Unix 工具打造很多只包含一行代码的调试程序，这种程序所遵循的流程大致是：获取——筛选——处理——汇总。我们可能需要运用管道机制来把各个部分连成整体。最有用的管道操作符就是管道符号（|），它可以把上一个步骤所输出的内容发送给下一个步骤，令其成为该步骤的输入值。

你需要处理的大部分数据，都是文本形式的数据，它们可以直接发送给 Unix 工具的标准输入端。如果数据不是文本形式，那就需要对其进行转换。例如，对于 Unix 系统、Windows 系统及 Java 平台的目标文件来说，可以分别使用 nm、dumpbin 及 javap 命令来深入查看其内容。例如，如果某个 C 或 C++ 程序意外地退出，那么可以用 nm 命令查看其目标文件，以获知其中哪些文件调用了（或者说引入了）exit 函数。

```
# List symbols in all object files prefixed by file name
nm -A *.o |
# List lines ending in U exit
grep 'U exit$'
```

该命令所输出的结果，可能要比搜索源代码所得到的结果更为精确，例如，它会输出下面这样的内容：

```
cscout.o:          U exit
error.o:          U exit
idquery.o:          U exit
md5.o:          U exit
pdtoken.o:          U exit
```

如果要处理的文件位于压缩文档中，那么可以用 tar、jar 或 ar 命令来查看压缩包的内容。如果要处理的数据位于一大批文件中，那么可以用 find 命令来寻找你所关注的内容。如果要处理的数据存放在网上，那么可以用 curl 或 wget 把它下载下来。此外，也可以通过 dd（结合 /dev/zero 这个特殊文件）、yes 以及 jot 命令来生成一些人工数据，在进行快速的 benchmark（基准测试）时，可能会用到这些命令。最后要注意，在处理由编译器所给出的一系列错误消息时，你可能想把标准错误重定向到标准输出或某个文件上面，这可以通过 2>&1 及 2>filename 等形式来实现。例如，在修改了某个函数的接口之后，你想把受到影响的所有文件全都编辑一遍，那么可以通过下面这条命令管道，把这些文件显示出来。

```
# Attempt to build all affected files redirecting standard error
# to standard output
make -k 2>&1 |
# Print name of file where the error occurred
awk -F: '/no matching function for call to Myclass::myFunc/
  { print $1}' |
# List each file only once
sort -u
```

对于日志文件以及调试所用的其他数据源来说，其中所包含的数据是多种多样的，因此在大多数情况下，我们都只需要用到其中的某一部分数据。我们有可能只需要处理每一行中的某些字段，或是只需要处理全部文本中的某些文本行。如果文本行包含宽度固定的字段，或是包含由空格或其他字段分隔符所区隔的元素，那么可以通过 cut 命令把某一列的内容裁剪出来。如果文本行无法清晰地分隔成字段，那么我们一般会通过带有正则表达式的 sed 命令，把自己所需的那一部分提取出来。

如果要从许多文本行中获取其中的一部分文本行，那么应该使用 grep 命令。我们通过正则表达式来指定待匹配的文本行，并通过 -v 标志来把不需要处理的文本行过滤掉。在第 21 条中，笔者曾经用下面这条管道命令来寻找包含除法操作但是除数不含 sizeof 的文本行。

```
grep -r ' / ' . |
grep -v '/ sizeof'
```

如果你要根据纯文字而不是正则表达式来进行搜索，而且待搜索的内容保存在文件中（该文件有可能是由前一个步骤所生成的），那么可以使用带有 -f 标志的 fgrep 来实现（该命令是一种适用于固定字符串的 grep 命令）。比较复杂一些的筛选标准，可以写成 awk 命令的模式表达式（pattern expression）。我们经常需要对这些办法进行多次组合，以获取自己想要的结果。例如，可以先用 grep 命令找出自己感兴趣的那些文本行，然后通过 grep-v 命令排除一些与问题无关的样本，最后通过 awk 命令把每一行中的特定字段提取出来。例如，下面这一系列命令，就能够对 trace.out 文件进行搜索，并显示出已经顺利开启的每一份文件的名字。

```
# Output lines that call open
grep '^open(' trace.out |
# Remove failed open calls (those that return -1)
grep -v '= -1' |
# Print the second field separated by quotes
awk -F\" '{print $2}'
```

（上面这些命令其实只需要用一条 awk 命令就可以实现，但是像这样分步骤来构建命令管道，做起来会更容易一些。）

处理数据的时候，我们经常需要根据特定的字段来对文本行进行排序。sort 命令有数十种选项，可以指定排序所用的键、这些键的类型以及输出的顺序。排好序之后，接下来可以统计每一种元素的出现次数，这可以通过带有 -c 选项的 uniq 命令来实现，然后，我们通常还需要对 uniq 命令所给出的结果再排一次顺序，这次采用的是 -n 标志，它能够按照数字顺序进行排列，使我们可以看到出现最为频繁的那个元素。有时我们需要对比某一个程序前后两次的运行结果。如果这两次运行的结果应该是相同的（这个运行结果或许指的是回归测试所输出的内容），那么就用 diff 命令来看看它们是不是真的完全一样，如果要对比的是两份有序列表，则应该使用 comm 命令。更为复杂一些的任务，还是需要用到 awk 命令。下面举一个例子。假设我们要调查资源泄漏的原因，那么首先应该做的事情，就是把那些虽然直接调用了 obtainResource，但却没有直接调用 releaseResource 的文件全都找出来。这可以用下面这条命令管道来实现。

```
# List records occurring only in the first set
comm -23 <(
# List names of files containing obtainResource
grep -rl obtainResource . | sort) <(
```

```
# List names of files containing releaseResource
grep -rl releaseResource . | sort)
```

（bash shell 提供了一种扩展机制，可以把一串命令以 <(...) 的形式括起来，使得这些命令的执行结果，能够像文件那样，作为参数传给其他命令。）

在很多情况下，处理完之后的数据量依然很大，还需要进行进一步的加工。比如，我们并不关心日志里面有哪些文本行用来表示程序错误，而是关心错误出现了多少次。其实有很多问题，都可以通过对前一个步骤的输出信息进行字数统计而解决，我们只需要使用简单的 wc（word count，字数统计）命令，并配以 –l 标志即可。如果想显示出结果列表中的开头 10 个或末尾 10 个元素，那么可以把该列表传给 head 或 tail 命令。下面这一串命令，可以找到对某份文件最为熟悉的那些人（在寻找代码评审员的时候，可能会用到这些命令）。

```
# List each line's last modification
git blame --line-porcelain Foo.java |
# Obtain the author
grep '^author ' |
# Sort to bring the same names together
sort |
# Count by number of each name's occurrences
uniq -c |
# Sort by number of occurrences
sort -rn |
# List the top ones
head
```

tail 命令尤其适合用来查看日志文件（参见第 23 条和第 56 条）。此外，如果要详细查看大量文本，那么可以经由管道，将其输送给 more 或 less 命令，这两条命令都允许我们在文本查看界面中来回浏览，并搜寻特定的字符串。如果这些方法都不能满足需要，我们可以考虑用 awk 命令来实现，比如，当我们想通过 sum+=$3 这样的形式，对特定字段进行求和的时候，通常就可以采用 awk 命令来做。下面这个例子会对 Web 服务器的日志进行处理，显示出请求的数量以及每次请求所传输的平均字节数。

```
awk '
# When the HTTP result code is success (200)
# sum field 10 (number of bytes transferred)
$9 == 200 {sum += $10; count++}
# When input finishes, print count and average
END {print count, sum / count}' /var/log/access.log
```

非卖品，仅供非商业用途或交流学习使用

各种 Unix 构建块，必须相互粘合起来，才能发挥效用。为此，我们需要使用 Bourne shell 的某些机制。例如，如果要用多个不同的参数来分别执行同一个程序，那么可以把这些参数传给 xargs 命令的输入端。典型的用法是通过 find 命令构造一份文件列表，然后用 xargs 命令对其进行处理。由于这种用法非常普遍，因此这两个命令都分别提供了相关的选项（-print0 及 –0），使得其数据以 null 字符而非空格符结尾，以便正确处理包含空格的文件名（例如，Windows 系统的“Program Files”文件夹，其名称中就带有空格）。假设要从创建时间晚于 foo.cpp 文件修改时间的那些日志文件里面，把字符串“access failure”出现次数最多的那份日志找出来，那么可以用下面这条命令管道来实现。

```
# Find all files in the /var/log/acme folder
# that were modified after changing foo.cpp
find /var/log/acme -type f -cnewer ~/src/acme/foo.cpp -print0 |
# Apply fgrep to count number of 'access failure' occurrences
xargs -0 fgrep -c 'access failure' |
# Sort the :-separated results in reverse numerical order
# according to the value of the second field
sort -t: -rn -k2 |
# Print the top result
head -1
```

如果要做的处理比较复杂，那么可以通过管道把数据发送给 while read 循环（Bourne shell 的功能很强大，我们可以通过管道把数据送入各种控制结构中，也可以通过管道将数据从控制结构中导出）。比如，如果你怀疑某个问题是由于更新了系统的 DLL（dynamically linked library，动态链接库）文件，那么可以用下面这条命令管道显示出 windows/system32 目录里面所有 DLL 文件的版本。

```
# Find all DLL files
find /cygdrive/c/Windows/system32 -type f -name \*.dll |
# For each file
while read f ; do
  # Obtain its Windows path with escaped \
  wname=$(cygpath -w $f | sed 's/\\/\\\\/g')
  # Run WMIC query to get its name and version
  wmic datafile where "Name=\"$wname\"" get name, version
done |
# Remove headers and blank lines
grep windows
```

如果在尝试了各种管道拼接的办法之后，还是无法达到想要的结果，那么可以用一系列中介文件来处理这些数据。

非卖品，仅供非商业用途或交流学习使用

要点

- 用 Unix 命令来获取、筛选、处理并汇总文本记录，从而实现对调试数据的分析。
- 把 Unix 命令用管道连接起来，可以迅速完成很多复杂的分析任务。

第 23 条：掌握命令行工具的各种选项及习惯用法

如果你要调试的程序打印出了“Missing foo”这样一条奇怪的错误消息，那么应该如何寻找造成该错误的源代码呢？这可以用下面的命令来实现：

```
fgrep -lr 'Missing foo' .
```

在包含应用程序源代码的目录里面运行上述命令，可以递归式地搜索（-r）其中的所有文件，并把包含错误消息的文件列出来（-l）。用 grep 命令来搜索文本，其优雅之处在于：无论产生错误消息的源代码是用哪一种编程语言写的，我们都可以把它找出来。如果应用程序使用了许多种编程语言，或是我们没有时间在 IDE 中为此专门设置一个项目，那么 grep 命令所提供的文本搜寻功能就显得尤为有用。请注意，fgrep 命令的 -r 选项，是 GNU 的扩展功能，崇尚经典 Unix 风格的人，可能不喜欢这种用法。如果你所用的系统没有这个功能，那么可以通过下列命令管道来实现同样的效果：

```
find . -type f |
xargs fgrep -l 'Missing foo'
```

我们要查看的数据中，通常会包含很多无关的信息，也就是说，会包含很多我们不想看到的内容。带有正则表达式的 grep 命令，可以把相关的信息筛选出来，然而在很多情况下，直接用 grep 命令的 -v 选项将无用的信息抛开，可能会更简单一些。许多条这样的命令组合起来之后，其效果特别好。比如，下面这条命令管道可以显示出包含字符串“Missing foo”，但是不包含“connection failure”或“test”的所有日志文件。

```
fgrep 'Missing foo' *.log |
fgrep -v 'connection failure' |
fgrep -v test
```

grep 命令可以把与正则表达式相匹配的所有文本行打印出来，但是如果这些文本行比较长，那么或许很难看出程序错误究竟是由其中哪一部分代码所造成的。比如，我们

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

怀疑某个问题可能与一份（格式不当的）HTML 文件中的 table 标记有关，那么怎样才能把这些标记（tag）迅速浏览一遍呢？这可以通过 grep 命令的 --color 选项来实现。比如，grep--color table file.html 命令会用红色凸现所有的 table 标签，以便于查看。

按照惯例，运行在命令行界面中的程序，不会把错误消息发送到标准输出端，因为那样做可能会对需要处理标准输出端的其他程序造成干扰，此外，如果程序的操作员把输出端重定向到某个文件中，那么还会造成该操作员很难在文件里面分辨出这些错误消息。因此，程序一般会把错误消息发送到另外一个渠道，也就是标准错误端。如果在终端界面中执行命令，那么即便对其输出做了重定向，我们通常也还是能够看到这些错误消息。然而在调试程序的时候，我们可能需要对错误消息进行处理，而不想看它们在屏幕上面飞滚，于是，可以用两种办法来对标准错误端进行重定向。第一种办法，是在运行程序的时候，通过 2>filename 这样的形式来把标准错误端（按照惯例，其文件描述符是 2）发送到某个文件中，以便进行后续的处理。还有一种办法，是把标准错误端重定向到与标准输出端相同的那个文件描述符上面（其文件描述符是 1），以便通过同一个管道来处理这两种信息。比如，下面这条命令可以把标准输出与标准错误一起发送给 more 命令，使得我们能够在其界面中随意翻看这些信息。

```
program 2>&1 | more
```

在调试 web 服务器等非交互式的程序时，我们所关注的那些操作，通常都会记录在日志文件中（参见第 56 条）。我们不应该频繁地打开日志文件，去查看里面发生了什么变化，而是应该用带有 -f 选项的 tail 命令来监测它的增长情况。该命令会一直打开日志文件，并且会注册一个事件处理程序。当日志发生变动时，这个处理程序就会得到通知，从而使 tail 命令能够及时展示日志中的变化。向日志文件中写入内容的那个进程，在某些情况下可能会删除日志或修改其名称，同时创建一份与原日志同名的新文件（比如，它可能是想把原来的日志内容保存成另外一个文件，并且用新的日志文件来记录新内容[⊖]），在这种情况下，我们可以给 tail 命令传入 --follow=name 选项，使得该命令能够按照文件名，而不是按照文件描述符来追踪文件。用 tail 命令监测日志文件的时候，应该单独把它放在一个窗口里面去观察（这个窗口可以调得小一些），这样就可以在调试应用程序的过程中，随时看到该文件所发生的变化了。如果日志里面有很多与调试工作无关的文本行，那么可以用 grep 命令对 tail 的输出信息进行搜索，从而把感兴趣的那部分内容展示出来。

⊖ 也称为日志轮替（logrotate）。——译者注

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
sudo tail /var/log/maillog | fgrep 'max connection rate'
```

如果要调试一种不经常出现的程序错误，那么应该设置一套监测机制，使我们能够在程序出现该错误时得到通知（参见第 27 条）。对于那些只需要监测一次的情况来说，可以给程序后面加上 & 符号，并且通过 nohup 工具来执行，这样，即便登出了当前的 shell，系统也依然会在后台运行该程序。稍后我们可以在名为 nohup.out 的文件里面查看程序的输出信息与错误信息。我们也可以通过管道，把程序的输出信息发送给 mail 命令，以便在程序执行完毕之后得到通知。对于在当天的工作时间之内就能够运行完的程序来说，可以通过下面这条命令，使得系统在执行完该程序之后发出提示音。

```
long-running-regression-test ; printf '\a'
```

把上面两种技巧结合起来，就可以在日志里面出现特定的文本行时，发出提示音或寄送邮件了。

```
sudo tail -f /var/log/secure |
fgrep -q 'Invalid user' ; printf '\a'

sudo tail -f /var/log/secure |
fgrep -m 1 'Invalid user' |
mail -s Intrusion jdh@example.com
```

如果修改上述命令，将其与 while read 循环相结合，那么还可以使提示的过程一直运行。不过这种做法实际上相当于在搭建底层的监控系统，对于这种用途来说，有更为专业的工具可供考虑（参见第 27 条）。

要点

- 通过 grep 命令的各种选项对搜索的结果进行逐步筛选。
- 对程序的标准错误端进行重定向，以便于分析。
- 用 tail-f 命令来监控内容持续增加的日志文件。

第 24 条：用编辑器对调试程序时所需的数据进行浏览

大家可能都认为调试器是调试工作的主要帮手，但是代码编辑器与 IDE，通常也可以很好地确定出 bug 的源头。这里所说的编辑器和 IDE，是指 Emacs 或 vim 等功能强大的编辑器与 IDE。无论你选用的是哪种编辑器，它都应该比系统自带的编辑器更为高

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

级，也就是说，要比 Windows 系统的 Notepad、OS X 系统的 TextEdit 以及各种 Unix 发行版的 Nano 与 Pico 更为高级，因为这些内置的编辑器仅仅提供了最基本的功能。

编辑器的搜索功能，可以在代码里面寻找与待查的问题有关的线索。IDE 的搜索功能可以把用到某个标识符的那些代码全都找出来，而编辑器的搜索功能则显得更为丰富，因为它可以用更加灵活的方式进行搜索，而且可以把出现在注释中的文字也纳入搜索范围。要想更加灵活地搜索，有一种办法是搜索词根而不搜索原词。比如，如果要寻找与排序问题有关的代码，那么应该搜索“order”而非“ordering”，因为在搜索 order 的同时，还可以找到 orders 和 ordering 等源自相同词根的词语。此外，还可以用正则表达式来囊括自己所关注的各种字符串，比如，如果待调试的问题涉及 x1、x2、y1 或 y2 这四个坐标字段，那么可以通过搜索 [xy][12] 来查找使用这四个字段的代码。

在其他一些情况下，编辑器可以帮我们找出代码为什么不能像预期的那样运作。比如，下面这段 JavaScript 代码，无法显示出本来应该展示的那条错误消息。

```
var failureMessage = "Failure!", failureOccurrances = 5;
// More code here
if (failureOccurrences > 0)
   alert(failureMessage);
```

上面这段代码的错误其实是很明显的，但是对于忙碌地调试了一整天了人来说，则未必能够发现这个小小的问题。不过，我们只需要用编辑器搜索代码里面的“failureOccurrences”，就可以立刻发现问题所在：代码里面本来应该有两个地方使用该变量，但是却只能找到一处，因为另外一处误写为“failureOccurrances”了。如果你在打变量名的时候，不是从当前这份源文件里面复制，而是通过其他手段来获取，那么这种搜索标识符的技巧，就显得尤为有用了，比如，你可能是从对该变量做出定义的那份文件，或是从某条错误消息里面复制出来的，也有可能自己照着这个单词，一个一个字母打上去的。编辑器能够把文件里面出现同一个词的地方全都找出来。对于 vim 编辑器来说，可以把光标放在待搜索的标识符那里，然后按下 * 键向前搜索，或按下 # 键向后搜索。Emacs 编辑器使用 Ctrl-s, Ctrl-w 来实现相同的功能。

如果想在调试的时候寻找两份文件中的不同点（参见第 5 条），那么编辑器是个很有用的工具。对于两个较为复杂的语句来说，可以把其中一个语句粘贴到另外一个的正下方，这样就可以通过逐个字符的比对来迅速判断出它们是否相同，而不用在屏幕上面来回地看了。如果要对比的是两块比较长的代码，那么可以把编辑器窗口分成左

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

右两部分，并且把其中一份代码放在另一份代码旁边，这样就可以看出二者之间的重要区别了。按道理来说，diff 这样的工具应该更适合用来寻找两份文件之间的区别，然而麻烦的地方在于：如果这两者之间的某些区别，体现在 IP 地址、时间戳或传给例程的参数等无关紧要的元素上面，那么 diff 所输出的结果就无法突出重点了。因此，在进行比较之前，应该先把这些无谓的因素去掉。比如，可以在 vim 中采用下面这个正则表达式把日志文件里面的 Chrome 版本号替换掉，令其只保留主版本号，例如，Chrome/45.0.2454.101 会变成 Chrome/45。

```
:%s/\(Chrome\/[^.]*\)[^ ]*/\1
```

如果要在含有大量数据的日志文件里面寻找错误原因，那么编辑器也可以帮上大忙。首先，可以用编辑器轻易地删掉那些与调试无关的内容。比如，如果想把日志文件中含有 poll 一词的文本行全都删掉，那么只需要在 vi 里面输入 :g/poll/d 就可以了，如果使用的是 Emacs 编辑器，那就先按下 M-x，然后执行 delete-matching-lines。这些命令可以反复执行，直到日志文件里面只剩下那些你所关注的内容（如果执行得过头了，可以通过“撤销”（undo）命令来复原）。如果删除完之后，日志文件的内容读起来还是太过复杂，那么可以在你所能够理解的地方加上“start of transaction”（事务开始）、“transaction failed”（事务失败）以及“retry”（重试）之类的注释。如果你要浏览包含逻辑结构块的大型文件，那么还可以通过编辑器的大纲（outlining，概要查看）功能，迅速地展开或折叠这些结构，并在各结构之间快速浏览。此时，你可以把编辑器窗口分成许多区域，以便同时查看彼此相关的多个部分。

要点

- 使用编辑器的搜索功能来寻找拼写有误的标识符。
- 对文本文件进行编辑，以突出其中的不同点。
- 对日志文件进行编辑，令其更加易读。

第 25 条：优化工作环境

调试是一件必须专心来完成的工作。如果你的开发环境配置得不到位，那么工作效率很可能会大幅降低。这本书有很多地方都在讲应该怎样高效地使用调试器（参见第 4

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

章）、编辑器（参见第 24 条）以及命令行工具（参见第 22 条）。除了这些内容之外，还有一些与工作效率有关的事情需要注意。

首先，要保证硬件与软件都合乎你的需要。在硬件方面，要确保你所使用的本地计算机或云端主机拥有合适的 CPU、内存及辅助存储空间。某些静态分析工具，要求计算机必须具备强劲的 CPU 及大量的内存，而对于其他一些任务来说，我们也有可能需要用大量磁盘空间来保存同一个开发项目的多份拷贝，以及很多 GB 的日志或远程测试数据。此外，如果能够在云平台上面轻易地启动多个主机实例，那么工作效率也可以得到提升。我们不应该为这些资源而纠结，因为和它们的开销相比，时间肯定要（或者说应该要）宝贵得多。软件方面也是如此。我们不应该受制于一些看似省钱而实际上却开销更高的做法，也不应该在自己所能下载、安装及使用的软件上面受到局限。如果某款软件能够帮你调试问题，那么没有理由不给你提供这款软件。调试工作本身已经够难的了，如果还要在设施与工具上面受到限制，那就会更加麻烦。

把这些资源汇聚起来之后，应该花一些工夫去配置它们，令其可以发挥出最大的效用。你应该根据自己的习惯，对按键绑定、别名、辅助脚本、快捷方式以及工具的配置进行优化，以便极大地提升调试效率。下面列出一些可以优化的地方，以及与之对应的 Bash 命令。

- 确保 PATH 环境变量中含有你所需的各种目录，使得你要运行的所有程序都能够涵盖在这些目录的范围之内。调试程序的时候通常会用到系统管理命令，因此应该把相关的目录包含在 PATH 变量中：

```
export PATH="/sbin:/usr/sbin:$PATH"
```

- 对 shell 和编辑器进行配置，使其可以推断出你要输入的内容，并自动把命令补齐。下面这个范例适用于 Git，它可以令你在各分支之间进行切换的时候，少敲很多键。

```
# Obtain a copy of the Git completion script
if ! [ -f ~/.bash_completion.d/git-completion.bash ] ; then
  mkdir -p ~/.bash_completion.d
  curl https://raw.githubusercontent.com/git/git/master/\
contrib/completion/git-completion.bash \
   >~/.bash_completion.d/git-completion.bash
fi
# Enable completion of Git commands
source ~/.bash_completion.d/git-completion.bash
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 对 shell 界面的命令提示符进行设置，使其可以显示出登录所用的身份、当前的目录以及主机的名称。调试的时候，经常要在多台主机与多个身份之间来回切换，因此把这些状态明确显示出来，有助于我们保持清醒。

```
# Primary prompt
PS1="[\u@\h \W]\\$ "
# Terminal bar
PROMPT_COMMAND='printf "\033]0;%s@%s:%s\007" "${USER}"\
"${HOSTNAME%%.*}" "${PWD/#$HOME/~}"'
```

- 把各种命令行编辑按键，设置得与你习惯的编辑器相符。如果你是通过逐渐拼接的方式来构建数据分析管道的，那么这样做将会提升工作效率（参见第 22 条）。

```
set -o emacs
# Or
set -o vi
```

- 为常用或是容易打错的命令设置别名或快捷方式。

```
alias h='history 15'
alias j=jobs
alias mroe=more
```

- 把与各种实用工具有关的环境变量设置好，比如，可以通过下面这几个变量来指定版本控制系统所使用的分页显示程序（paging program）以及编辑器。

```
export PAGER=less
export VISUAL=vim
export EDITOR=ex
```

- 把输入的命令都记录到历史文件里面，这样就可以在几个月之后，搜索到当时所执行的调试命令了，这些历史信息是很有价值的。请注意，如果不想使某条命令记入历史文件（如包含密码的命令），那么可以在其前方加上空格。

```
# Increase history file size
export HISTFILESIZE=1000000000
export HISTSIZE=1000000
export HISTTIMEFORMAT="%F %T "
# Ignore duplicate lines and lines that start with space
export HISTCONTROL=ignoreboth
# Save multi line commands as single line with semicolons
shopt -s cmdhist
# Append to the history file
shopt -s histappend
```

非卖品，仅供非商业用途或交流学习使用

- 使 shell 的路径名扩展机制（也就是 globbing 机制，如对 * 进行扩展）能够涵盖子目录中的文件。

```
shopt -s globstar
```

开启上述选项之后，就可以通过 ** 通配符来对层次较深的目录结构执行命令了，这个通配符会扩展成该目录结构中的全部文件。比如，下面这条命令会对整个目录结构中的每一份 Java 源代码进行搜寻，并通过其 JavaDoc 标记来判断作者中有没有 James Gosling 的名字，然后把符合条件的文件数量显示出来。

```
$ grep '@author.*James Gosling' **/*.java | wc -l
33
```

配置完 shell 之后，还要配置其他的程序。你应该花时间去学习调试器、编辑器、IDE、版本控制系统以及分页显示程序的配置方式，把它们配置得与自己的习惯及工作风格相符。IDE 和复杂一些的编辑器，都会提供很多实用的插件，你应该把对自己有用的那些插件选出来，并制定一套简单的办法，使自己可以把这些插件安装在每一台需要用到的计算机上面。如果你能坚持采用这套办法进行配置；那么节省下来的时间，就会超过当初学习它们时所付出的时间。

调试过程中，通常需要同时操作多台计算机。在这种情境之下，有三项重要的技巧可以帮助我们节省时间。首先，要设法使自己无需输入密码，即可登录想要使用（或是在那里执行命令）的远程主机。对于 Unix 系统来说，只需设置好公钥—私钥对（这一般是通过 ssh-keygen 命令实现的），并把公钥放在远程主机里面名为 .ssh/authorized_hosts 的那个文件中即可。第二项技巧，是给主机设置既简短又好记的别名，这样就不用输入完整的主机名称了（此外，还可以设置采用该别名进行登录时所使用的用户名）。这些别名存放在 .ssh/config 文件里。比如，下面这几行代码，令我们可以通过 garfield 这个别名来以 testuser 的身份，登录名为 garfield.dev.asia.example.com 的 ssh 主机。

```
Host garfield
HostName garfield.dev.asia.example.com
User testuser
```

第三项技巧，是设法寻找能够从桌面直接登录远程主机，并在其中执行 GUI 应用程序的方式。尽管这种方式设置起来或许有一点麻烦，但在当前的大多数操作系统上面，都是可以实现的。如果能够从桌面上迅速地运行远程主机中的 GUI 调试器或 IDE，那么你的工作效率将会大幅提升。

非卖品，仅供非商业用途或交流学习使用

调试的时候，我们经常要同时操作命令行与 GUI 这两种界面。因此，为了节省时间，我们一定要知道应该怎样把这两种环境联系起来。有一项很有用的技巧，就是从命令行界面里启动 GUI 程序，比如，可以从命令行里启动待调试的应用程序，并指定一份你所构造的测试文件。Windows 系统使用 start 命令来启动，OS X 系统使用 open 命令来启动，Gnome 和 KDE 桌面环境分别采用 gnome-open 及 kde-open 来启动。我们还可以设法在命令行与 GUI 的剪贴板之间复制文字，比如，把内存转储文件所在的那个很长的路径复制下来。Windows 系统可以用 Outwit 套件中的 winclip 命令来实现，如果安装了 Cygwin，那么也可以对 /dev/clipboard 文件进行读取或写入。在 Gnome 和 KDE 桌面环境下，可以使用 xsel 命令来实现。如果我们使用多种 GUI 环境，那么可以创建一个命令别名，使它在各种环境之下，都能够适配到正确的命令上面。比如，下面这几行代码可以使我们通过“s”命令，以 GUI 形式开启文件或执行 GUI 应用程序。

```
case $(uname) in
FreeBSD) # KDE
  alias s=kde-open
  ;;
Linux) # Gnome
  alias s=gnome-open
  ;;
Darwin)
  alias s=open
  ;;
CYGWIN*)
  alias s=cygstart
  ;;
esac
```

我们还可以对 GUI 环境进行配置，使自己能够从文件的关联菜单[⊖]中启动编辑器，并且能够从文件夹的关联菜单中，开启以该文件夹为当前路径的 shell 窗口。另外还有一个技巧不知你是否了解：GUI 文件浏览器里面的文件名，是能够拖放到 shell 窗口中的。你可以试一下，效果很棒。

把上面这些内容精心调配好之后，我们应该设法确保这套配置能够连贯地套用在调试软件时所用的各台主机上面。有一个很巧妙的办法，就是把这些配置文件纳入版本控制系统中。这样，如果将来我们在某台计算机上面对配置文件做了改进，或是修复了其中的兼容问题，那么只需要把它们推送到中心存储库里面就行了，稍后在使用其他计算

⊖ context menu，也称为上下文菜单、快捷菜单或弹出式菜单。一般可以通过在物件上面右击而触发。——译者注

非卖品，仅供非商业用途或交流学习使用

机时，可以把这些文件拉取出来。如果是在一台新计算机上面工作，那就直接把整个存储库里面的文件签出到 home 目录。用 Git 来管理配置文件的时候，可以通过 .gitignore 文件来指明 home 目录里面有哪些内容应该纳入管理范围。比如：

```
# Ignore everything
*
# But not these files...
!.bashrc
!.editrc
!.gdbinit
!.gitconfig
!.gitignore
!.inputrc
```

请注意，上面所给出的那些建议，是笔者根据自己多年的经验而提出的。你所面对的需求与开发环境，或许与笔者有很大区别。你应该经常观察自己的开发环境中有哪些用起来不太顺手的地方，并设法解决这些问题。如果你总是发现自己需要输入一长串命令或是单击很多次才能执行某项操作，那就看看能不能通过脚本将其自动化。如果你发现某些工具总是帮倒忙，那就看看怎样才能把它们配置得满意一些，或是看看能不能改用更好的工具。最后，要多向其他人打听各种技巧及工具，因为别人可能已经找到了一种优雅的方案，可以用来解决某个困扰着你的问题。

要点

- 适当地配置自己所用的工具，以提升工作效率。
- 通过版本控制系统，在各台计算机之间共用同一套环境配置方案。

第 26 条：用版本控制系统寻找 bug 发生的原因及经过

我们所遇到的很多 bug，都与软件的改动有关。在添加了新的特性并做出了新的修正之后，软件里面难免会有新的 bug 出现。Git、Mercurial、Subversion 或 CVS 这样的修订控制系统，可以使我们详细查看软件的修改记录，并从中获取与当前问题有关的重要线索。为了发挥这一优势，我们必须把自己对软件所做的修改，用心地记录到版本控制系统中（参见第 10 条）。所谓用心，是指每一次修改都应该单独提交，而且要写上有意义的提交消息，如果有可能，还应该链接到对应的事务上面（参见第 1 条）。

现在给出一些范例，以演示怎样通过版本控制系统来辅助我们的调试工作。这些范

例是用 Git 的命令行操作来撰写的，因为这种方式适用于各种环境。如果你喜欢用 GUI 工具来执行这些操作，那么完全可以继续使用那个工具。如果你使用的是另外一种版本控制系统，那么请查阅其文档，看看怎样执行对应的操作，或者也可以考虑改用 Git，因为它提供了很多强大的功能。请注意，并非每一种版本控制系统都是“生而平等”的，有一些系统对本地的分支及合并操作支持得很糟糕，而这两种操作对调试工作来说，却又是相当重要的，因为我们经常要尝试好几种不同的实现方法。

当我们发现新 bug 的时候，首先应该查看软件所经历的变化。

```
git log
```

如果你知道该问题与某个具体的文件有关，那就在命令里面指定这个文件，只查看涉及该文件的变化。

```
git log path/to/myfile.js
```

如果你怀疑问题与某几行特定的代码有关，那么可以获取一份写有注解的代码清单，其中的每行代码旁边，都会标有与上次修改有关的细节。

```
git blame path/to/myfile.js
```

（可以指定 -C 及 -M 选项，以展示文件内部及文件之间的代码行移动情况。）

如果与问题有关的代码已经不在当前这个版本里面了，那么可以在过去的版本中搜寻指定的字符串。

```
git rev-list --all | xargs git grep extinctMethodName
```

如果我们知道问题是从某一个版本之后才开始发生的（如 V1.2.3），那么可以只看发生在该版本之后的变化。

```
git log V1.2.3..
```

如果不知道版本号，但是知道出现问题的日期，那么可以获取在该日之前最后一次提交所对应的 SHA hash。

```
git rev-list -n 1 --before=2015-08-01 master
```

然后，可以把这个 SHA hash 放在本来应该传入版本号的地方。

如果我们知道问题是在解决了某个事务之后出现的（如第 1234 号事务），那么可以

搜索与该事务有关的提交。

```
git log --all --grep='Issue #1234'
```

（我们在进行与事务有关的提交操作时，必须在消息里面以“Issue#1234”的形式标出这个事务，才能使刚才的那条命令见效。）

在用上述各种命名来追溯代码的变化情况时，只要你知道与某次提交相对应的那个 SHA hash（如 1cb6e3f6），那么就可以通过指定该 hash 来展示与之有关的修改。

```
git show 1cb6e3f6
```

下面这条命令可以显示出两个版本之间所经历的变化。

```
git diff V1.2.3..V1.3.2
```

一般来说，只要查看一下代码的变化情况，就能够找到问题的原因了。我们也可以根据提交信息中所写的名字，找到与某次修改有关的开发者，询问他们在修改代码时所依循的思路。

版本控制系统还可以当成时光机来用。比如，可以把原来某个正确的版本（如 V1.1.0）签出，将其代码放在调试器中运行，并与当前的版本相互比对（参见第 5 条）。

```
git checkout V1.1.0
```

还有一种很有特点的用法。如果你知道 bug 是由某两个版本之间（如 V1.1.0 和 V1.2.3 之间）的某个版本所引入的，而且有一个能够在测试失败时返回非 0 值的脚本（如 test.sh），那么就可以用 Git 对这个范围内的所有版本进行二分查找，以确定引入 bug 的那个版本。

```
git bisect start V1.1.0 V1.2.3
git bisect run test.sh
git reset
```

我们可以用 Git 创建本地分支，并在上面进行实验，以求修复某个问题。如果修好了，那就将该分支与 master 合并，若是修不好，则将其删除。

```
# If the experiment was successful integrate the branch
git checkout master
git merge issue-work-1234

# If the experiment failed delete the branch
git checkout master
git checkout -D issue-work-1234
```

如果你正在做某件事，公司突然叫你去解决另外一个事务，那么可以把当前的变更先隐藏起来，等到处理完客户的那个问题之后，再将其恢复。

```
git stash save interrupted-to-work-on-V1234
# Work on the debugging issue
git stash pop
```

要点

- 用版本控制系统来查看文件的修订记录，以确定 bug 是在什么时候、以何种方式引入的。
- 用版本控制系统来查看正常运行的版本与出现故障的版本之间有何区别。

第 27 条：用工具监测由多个独立程序所构成的系统

如果软件系统里面只有一个程序，那么软件出错的时候，只需要调试这个程序就可以了，然而当前很少有哪个基于软件的系统，是仅由一个程序所构成的，它们几乎都包含着各种各样的服务、组件及程序库。因此，我们应该迅速而高效地找到发生故障的那个元素，以便在调试此类系统的时候占得先机。要想做到这一点，只需要在服务器端使用或者配置一套基础设施监测系统，并将其运行起来就可以了。

下面我们将以流行的 Nagios 工具为例来进行讲解。该工具可以作为自由软件来使用，也可以当作一款商业软件来购买，厂家会给购买者提供产品与服务支持。如果你的公司使用的是另外一套监测系统，那么也可以继续使用那套系统，因为原理都是相同的。无论选用哪种产品，都不要急着去自己构建这样的监测系统。与自己快速打造出来的系统或是像 collectd 和 RRDtool 这样的被动记录系统（passive recording system）相比，Nagios 有很多优势，例如，具有经过测试的服务检查程序及通知程序（被动与主动的都有）、管理面板、轮询事件数据库、非侵入式的监测计划安排功能、可扩展性以及庞大的用户社群（他们会给 Nagios 开发插件）。

如果你开发的软件要运行在云端主机上面，或是要基于某种常见的应用程序栈，那么厂商或许会以服务的形式，提供一套基于云端的监控系统给你使用。例如，Amazon Web Services（AWS）就提供了监测机制，可以对其所提供的服务进行监控。

为了能够有效地探查问题，我们必须对应用程序进行全方位的监测。首先从最底层

的资源开始，也就是要监测每一台主机的健康状况，其中包括：CPU 的负载、内存的用量、网络是否可达、正在执行的进程数量、已经登录的用户数量、可以更新的软件、剩余的磁盘空间、已经打开的文件描述符、已经占用的网络带宽与磁盘带宽、系统日志、安全性以及远程访问。然后我们向上走一层，看看软件所需的那些服务，能不能够正确而可靠地运行。需要监测的地方有：数据库、电子邮件服务器、应用程序服务器、缓存、网络连接、备份、队列、消息传递、软件授权、Web 服务器以及目录。最后，要对应用程序的健康状况进行详细监测。具体的细节可能会有所不同，然而下面几点最好能够监测到：

- 应用程序能否正确地处理整个流程（例如，从填写 Web 表单到完成交易的全过程）。
- 应用程序中的各个部分是否正常，例如，Web 服务、数据库表格、静态网页、交互式 Web 表单以及报告机制。
- 某些关键指标是否正常，例如，响应延迟、已经加入队列和已经完成的订单、活跃的用户数量、失败的交易、发生的错误以及得到报告的程序故障等。

软件发生故障时，Nagios 会在其 Web 界面中更新相关服务的状态。通过图 3.1，可以看到运行在各种主机上面的各项服务都处在何种状态。黄色表示警告，红色表示错误。此外，我们还可以选择在软件发生故障时，立刻收到短信或电子邮件通知。对于偶发的故障来说，这种通知可以令我们对出现故障的服务进行及时的调试，从而更快地找到问题的原因。我们也可以对 Nagios 进行设置，令其开启一张工单，使得我们能够对事务进行指派、追踪及归档（参见第 1 条）。Nagios 还可以把事件与服务之间随着时间的推移所表现出来的关系，绘制成直方图。仔细观察故障的发生时间，可以帮助我们看出与故障有关的其他一些因素，如 CPU 负载过高或内存压力过大等。如果我们是对服务所在的整个体系进行监测，那么就有可能发现：某些底层的错误会产生连锁反应，从而导致其他一系列问题。在这种情况下，我们通常应该首先查清引发错误的那个底层元素（参见第 4 条）。

如果 Nagios 提供的通知方式不符合你的需要，那么可以自己编写通知处理程序。程序清单 3.1 中的 shell 脚本使用了由 Stephen Celis 所编写的 ghi 工具，它使得 Nagios 能够在服务发生故障时，向 GitHub 提交一项事务。

程序清单 3.1　Nagios 插件范例

```
#!/bin/sh
TITLE="$1"
```

非卖品，仅供非商业用途或交流学习使用

```
BODY="$2"

# Unescape newlines
NLBODY="$(printf '%b' \"$BODY\")"
ghi open -m "$TITLE
$NLBODY
" >/dev/null
```

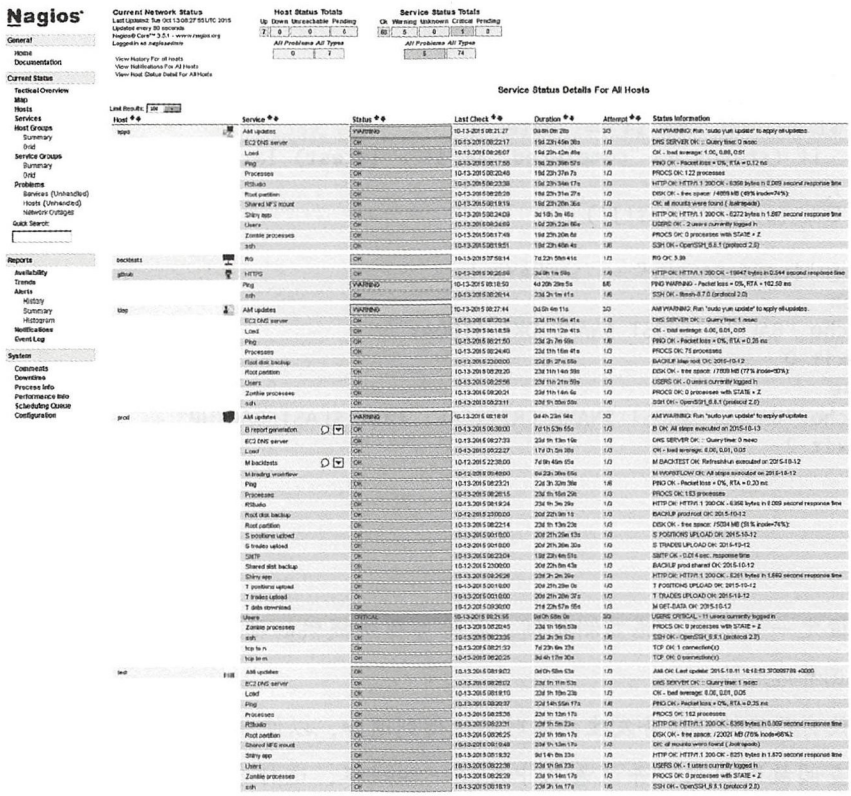


图 3.1　用 Nagios 监测服务的详细状态

Nagios 设置起来很简单。大多数流行的操作系统都提供了安装包，这种安装包内置了对关键的主机资源及常见的网络服务进行监控的功能。此外，还有一千多种插件，可以用来监测云端、集群、CMS、安全以及 Web 表单等各种各样的服务。如果这些插件不能够满足你的需求，那么可以用脚本来编写自己的检查程序。这样的脚本应该把服务的状态打印出来，并在退出的时候返回 0 或 2，以表明该服务是正常运行，还是出现了严重错误。下面这段 shell 脚本用来验证指定的存储卷（storage volume）是否已经备

非卖品，仅供非商业用途或交流学习使用

份为带有时间戳的 AWS 快照。

```
#!/bin/sh

HOST="$1"
NAME="$2"
TODAY=$(date -I)

LAST_BACKUP=$(ec2-describe-snapshots --filter \
   tag:Name="backup-$HOST-$NAME" \
   --filter tag-key=date |
  awk '
   $1 == "SNAPSHOT" {status = $4}
   $1 == "TAG" && $4 == "date" {
    if (status == "completed" && $5 > latest) latest = $5
   }
   END {print latest}')

if [ "$LAST_BACKUP" = "$TODAY" ]
then
  echo "BACKUP $HOST $NAME OK: $TODAY"
  exit 0
else
  echo "BACKUP $HOST $NAME CRITICAL: last $LAST_BACKUP"
  exit 2
fi
```

要点

- 设定一套基础设施监测机制，以检查你所提供的服务中的各个部分，是否都在正常地运作。
- 使自己能够在服务发生故障时迅速得到通知，以便在该状态下调试系统。
- 查阅故障记录，并试着从中发现一些规律，这样或许能够帮助你找到问题的原因。

非卖品，仅供非商业用途或交流学习使用

第 4 章 Chapter 4

调试器的使用技巧

调试器是一种专门的工具，用来详细检视软件在运行时的行为，它或许是最为任性的一类软件，因为没有别的软件可以像它这样，仅仅因为自身的用法，就要求 CPU 与操作系统都为其提供特殊的支持。由此可知，专业人士对它是十分重视的，并且会投入精力来做好这类软件。如果我们不能学会有效地使用调试器，那就无法发挥它的优势。由于每个人所使用的编程语言和开发环境各有不同，因此本章所讲的一些技巧对你来说，可能已经不再新鲜了。如果是这样，那么把这些技巧浏览一遍就可以了；否则，就请仔细阅读这些内容。

有很多独立的调试器可供我们选用，而且许多集成开发环境（IDE）也都支持调试功能，使用这些工具来调试软件时所遵循的方法与技巧，大致是相同的。本章选用三种流行的调试环境来演示怎样对编译好的代码进行调试，这三种环境是：调试 Java 与 Scala 代码的 Eclipse IDE，调试 C、C++、Visual C# 及 Visual Basic 等语言的 Visual Studio IDE，以及在 Unix 系统下调试 C、C++、D、Go、Objective-C、Fortran、Java、OpenCL C、Pascal、汇编、Modula-2 及 Ada 等语言的 gdb。如果你使用的是其他工具，如是用来调试 JavaScript 代码的 Google Chrome 浏览器，那么请查阅资料，看看怎样在那些工具里面运用本章所说的这些技巧。如果某个工具不支持你想要使用的功能，那么可以考虑改用其他工具。

非卖品，仅供非商业用途或交流学习使用

第 28 条：编译代码时把符号信息包含进来，以便于调试

尽管调试器可以用来调试任何一款编译好的程序，但是如果想完全发挥出它的优势，那么最好调试那种含有调试信息程序。这些信息可以把机器指令与源代码对应起来，并且可以把内存地址与变量对应起来。对于 C、C++ 及 Go 等编译成硬件机器指令的语言来说，应用程序只需要包含极少量的信息就可以运行，这些信息用来指出该程序在运行的时候，需要和哪些程序库相链接，这样的程序库，在 Unix 系统中叫做共享库，在 Windows 系统中称为动态链接库（dynamically linked library，DLL）。可是这样一来，就意味着调试器只能识别出程序所使用的少数几个例程。下面来看一个例子，这是一款 C++ 程序在等待输入时的调用栈（参见第 32 条）。

```
#0 0xb77c0424 in __kernel_vsyscall ()
#1 0xb75e7663 in read () from /lib/i386-linux-gnu/i686/cmov/
   libc.so.6
#2 0xb758bb3b in _IO_file_underflow ()
  from /lib/i386-linux-gnu/i686/cmov/libc.so.6
#3 0xb758d3db in _IO_default_uflow ()
  from /lib/i386-linux-gnu/i686/cmov/libc.so.6
#4 0xb758e808 in __uflow () from /lib/i386-linux-gnu/i686/
  cmov/libc.so.6
#5 0xb75840ec in getc () from /lib/i386-linux-gnu/i686/cmov/
  libc.so.6
#6 0xb7750345 in __gnu_cxx::stdio_sync_filebuf<char, std::
  char_traits<char> >::uflow() () from /usr/lib/i386-linux
  -gnu/libstdc++.so.6
#7 0xb7737365 in ?? () from /usr/lib/i386-linux-gnu/
  libstdc++.so.6
#8 0xb77386a4 in std::istream::get(char&) ()
  from /usr/lib/i386-linux-gnu/libstdc++.so.6
#9 0x0805083d in ?? ()
#10 0x0804f730 in ?? ()
#11 0x0804b792 in ?? ()
#12 0x08049b13 in ?? ()
#13 0x08049caf in ?? ()
#14 0xb7535e46 in __libc_start_main ()
  from /lib/i386-linux-gnu/i686/cmov/libc.so.6
#15 0x08049a01 in ?? ()
```

能够显示出名称的例程（如 get 和 read），是那些位于共享库中的例程，这个 C++ 程序必须与相应的共享库相链接，才能够正常地运行。（其名称以下划线开头的例程，是内部例程，它们也需要进行动态链接。）如果某个例程的名称显示为问号（??），那说明调试器无法将其与源代码对应起来。

非卖品，仅供非商业用途或交流学习使用

在 Unix 系统中编译程序时，默认会把那些将在链接阶段得以解析的变量名称与例程名称（称为非静态符号）包含进来。这些内容通常可以用名为 strip 的命令来移除，以便节省空间或掩盖专属信息。如果程序里面的这些内容没有遭到剥离，那么可以给我们提供更多的指示。下面我们来看看同样一个程序，在这些内容未遭移除的情况下，会具备怎样的调用栈。

```
#8 0xb769e6a4 in std::istream::get(char&) ()
  from /usr/lib/i386-linux-gnu/libstdc++.so.6
#9 0x0805083d in CMetricsCalculator::calculate_metrics_
   switch() ()
#10 0x0804f730 in CMetricsCalculator::calculate_metrics_
   loop() ()
#11 0x0804b792 in CMetricsCalculator::calculate_metrics() ()
#12 0x08049b13 in process_metrics(char const*) ()
#13 0x08049caf in main ()
```

我们可以看到，刚才显示为问号的地方，现在已经有具体的函数名了。file 命令能够指出 Unix 程序的符号信息是否已经移除。把符号信息已经移除和尚未移除的两个程序传给该命令，会分别产生下列输出。

```
/bin/bash: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.35, stripped

nethogs: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.18, not stripped
```

无论在哪个操作系统里面做开发，都可以给编译器和链接器设置一些选项，令其把更多的调试信息包含在编译好的程序中，这些信息包括指令的内存地址所对应的文件名与代码行的编号，以及各种变量的位置。如果我们在编译刚才那个 C++ 程序的时候，把这些信息也包含进来，那么其调用栈就会变成下面这个样子：

```
#9 0x0804f054 in CharSource::get (this=0xbfa0876c,
   c=@0xbfa08637: 8 '\b') at CharSource.h:32
#10 0x0804fa4d in CMetricsCalculator::calculate_metrics_switch
   (this=0xbfa0876c) at CMetricsCalculator.cpp:160
#11 0x0804eea5 in CMetricsCalculator::calculate_metrics_loop
   (this=0xbfa0876c) at CMetricsCalculator.cpp:868
#12 0x0804b4b2 in CMetricsCalculator::calculate_metrics
   (this=0xbfa0876c) at CMetricsCalculator.h:74
#13 0x08049833 in process_metrics (filename=0x8054dd8 "-")
   at qmcalc.cpp:39
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
#14 0x080499cf in main (argc=1, argv=0xbfa08b94)
   at qmcalc.cpp:72
```

现在可以看到每一次方法调用所对应的文件名及行号，而且还能看到参数的名称与值。这些数据正是我们在调试的时候需要用到的。不过要注意，在把程序发布给客户的时候，可能要删掉这些数据，以免泄漏公司的专有信息。

在程序中嵌入调试信息的具体办法，要根据你所使用的工具而定。下面以几种常见工具为例来说明与之相应的办法。

- 用 Eclipse 来开发 Java 程序的时候，它默认就会生成调试符号。你也可以通过 Project-Properties-Java Compiler-Classfile 来调整。
- Oracle JDK 的编译器提供了 -g 选项，可以把所有调试信息都嵌入程序中（该选项还有几个参数可以调整）。
- 大多数 Unix 编译器都采用 -g 选项来嵌入调试信息。
- Microsoft 的编译器采用 /Zi 选项来嵌入调试信息。
- 用 Microsoft 的 Visual Studio 做开发时，可以通过 Build-Configuration manager-Active solution configuration 来选择自己是要构建调试版的程序，还是要构建发行版的程序。（工具栏上也有与之对应的选项。）

如果程序是用来调试的，那么还应该注意在构建该程序时所采用的优化级别。当前有很多编译器都会进行大量的优化，这些优化会大幅修改生成后的代码，令你无法理解它们与自己所写的源代码之间的对应关系。如果你对这样的程序进行单步调试（参见第 29 条），那么调试器可能会表现得较为错乱、可能会跳过某些语句，也有可能会按照另外一种你意想不到的顺序来执行。为了演示该问题，我们来看下面这个 C 语言程序。

```
#include <stdio.h>
int main()
{
      int a, b, c, d, i;

      a = 12;
      b = 3;
      c = a / b;
      d = 0;

      for (i = 0; i < 10; i++)
            d += c;
      printf("result=%d\n", d);
}
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

如果用 Microsoft 的 C 语言编译器进行编译，并且把优化级别调到最大，那么就会生成下列代码：

```
$SG2799 DB  'result=%d', 0aH, 00H
_main PROC
      push  40
      push  OFFSET $SG2799
      call  _printf
      add   esp, 8
      xor   eax, eax
      ret   0
_main ENDP
```

它所生成的这段代码，只会把变量 d 的最终结果（也就是 40）推到栈中，并调用 printf 将其显示出来。这样做是正确的，这段代码并没有执行那些赋值、计算和循环等语句，因为编译器在编译的时候，就已经把这些计算做好了，于是在生成的代码里面，只需把预先算好的结果显示出来即可。然而，对于正在调试复杂算法的人来说，他是不希望看到某些代码就这样消失的。

因此，如果是为了调试而构建程序，那么请禁用编译器的优化功能。使用 IDE 做开发的时候，只需要在 debug 版本与 release 版本之间切换就可以了，IDE 会自动把优化选项控制好。Oracle 的 Java 编译器，不会做出优化，而是会把优化过程留给即时（just-in-time，JIT）编译器来做，后者是在运行 Java 程序的时候执行的，它会把 JVM 代码转换成机器码。从命令行界面编译代码时，需要通过命令选项来禁用编译器的优化功能，Unix 系统上面的选项是 -O0，Microsoft 编译工具的选项是 /Od。请注意，即便关掉优化功能，编译器也依然有可能会对程序做出修改，例如，可能会移除那些永远执行不到的代码（dead code），并且会对常量表达式提前进行求值等。构建 debug 版的程序时，通常应该关掉优化功能，但是构建 release 版的程序时，则不应该关掉这些功能，因为这会令生成出来的代码执行得很慢。有些公司为了使产品调试起来更容易，会在生成 release 版的程序时，也把优化功能关掉，这样还可以防止编译器做出不成熟或不正确的优化。还有一些公司为了便于调试，甚至会把调试符号也放到代码里面，然而这样做有一个风险，就是会使得别人更容易对其产品进行逆向工程。

在调试的时候，有两种情况是不应该关闭优化功能的。第一种情况是，程序中的某一部分代码对性能要求很高，而且要调试的 bug 并不在这一部分代码里面，因此，我们想令编译器对这一部分代码进行优化。在这种情况下，我们应该要根据自己的构建需求，对编译器的优化选项进行调整，好在很多编译器都容许经过优化的代码与未经优化

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

的代码共存。第二种情况是，我们遇到了只会在 release 版本里面出现的 bug，这种 bug 在 debug 版的程序里面看不到。在这种情况下，我们需要对构建程序所用的选项进行调整，使得编译器既要像构建 release 版本时那样，对程序进行优化，又要保留调试所需的信息。

要点

- 对构建程序所用的配置选项进行调整，使得调试信息的详细程度与你的需要相符。
- 禁用编译器的代码优化功能，以便使生成出来的代码能够与你所要调试的代码相对应。

第 29 条：对代码进行单步调试

想要对程序的详细执行情况进行实时追踪，是不可能的，因为计算机每秒钟要处理数十亿条指令。不过，我们可以用调试器的单步执行功能来运行程序，这样就可以一次只执行一条语句或一个机器指令了。在单步调试的过程中，程序会依次执行相关的指令，从而使我们能够找到问题的原因。比如，我们会发现某个条件判断语句选择了错误的分支，或是某个循环的执行次数多于或少于预期的次数等。在本章的其他条目中，我们还可以看到：单步调试机制，能够把程序在执行每行代码之前和之后的详细状态显示出来。

怎样开始进行单步调试，取决于你所使用的工具。用 Eclipse 做开发的时候，可以通过 Window-Open Perspective-Debug 菜单打开 debug perspective，然后通过 Run-Debug 菜单或 F6 键来运行程序，然后，就可以通过 F5 键来单步执行每一行语句了。用 Visual Studio 做开发的时候，可以通过 Debug-Step Into 或 F11 键来进行调试。在 Unix 命令行界面中，可以把程序的可执行文件作为参数传给 gdb（如 gdb path/to/myprog），然后在程序的入口处设置断点，比如，可以通过 break main 命令来在 main 那里设置断点（参见第 30 条）。接下来，就可以通过 step 命令进行单步调试了（其后也可以按下 Enter 键，以便反复下达该命令）。请注意，有很多面向对象的编程语言，在进入程序的 main 方法之前，会先通过运行静态对象的构造器来执行一些代码。

调试器通常会进入一些你并不想要深入调试的代码中。比如，在大多数情况下，

非卖品，仅供非商业用途或交流学习使用

你可能都不想深入查看循环中每一个例程的详细执行过程。调试器的 step over 命令（Eclipse：F6、Visual Studio：F10、gdb：next），可以用来应对这种情况。调试算法的时候，我们会用 step over 的方式，调试某些不太重要的例程，并且用 step into 的方式来深入调试那些有问题的例程。如果你发现自己不小心进入了某个例程，那么可以命令调试器直接运行到该例程的返回点（Eclipse：F7、Visual Studio：Shift-F11、gdb：finish）。

有的时候，我们在已经走过了某个例程之后，才发现自己其实应该进入那个例程中进行调试。这时我们可以给调用该例程的那条语句加上断点，并通过程序的用户界面，把有问题的部分重新执行一遍，或是把整个程序重新执行一遍。重新执行程序的时候，它就会停在即将调用该例程的断点那里，使我们有机会进入其中进行单步调试。这样的过程，令我们能够逐步缩减问题的排查范围，也就是说，我们总是先执行程序中的许多代码，直到碰见一个有问题的例程；然后，给调用该例程的那个地方加上断点，并重新执行程序；这一次，就可以进入那个有问题的例程了，接下来，重复刚才的做法，直至找到问题。此外，也可以考虑使用调试器的反向调试功能（参见第 31 条）。

要点

- 通过单步调试来查看语句的执行顺序及程序的状态。
- 为了提升调试速度，我们可以直接经过某些与 bug 无关的部分，而不用进入其中。
- 如果发现程序所经过的某个例程有问题，那就给该例程设置断点，重新运行程序，并进入例程中进行单步调试，以求缩小有待排查的范围。

第 30 条：设置代码断点和数据断点

代码断点可以令我们在调试的过程中，详细控制自己所要查看的代码。我们通过指定源代码中的位置来设置断点，使得程序在执行到这里之后就停下来，并把控制权交给调试器（Eclipse：Run-Toggle Breakpoint 或 Ctrl-Shift-B、Visual Studio：Debug-Toggle Breakpoint、gdb：break file-name:line-number）。除了通过指定源代码中的位置来设置断点，我们还可以通过例程的名称来设置断点（Eclipse：Run-Toggle Method Breakpoint、Visual Studio：Debug-New Breakpoint-Break at Function、gdb：break routine-name）。然后，我们可以通过 run 命令，从头运行程序（Eclipse：Run-Run 或 Ctrl-F11、Visual Studio：

非卖品，仅供非商业用途或交流学习使用

Debug-Start Debugging 或 F5、gdb ： run），或是通过 continue 命令，使得当前正在运行的程序继续往下执行（Eclipse ： Run-Resume 或 F8、Visual Studio ： Debug-Continue 或 F5、gdb：continue）。

断点使我们能够迅速把注意力放在那些有问题的代码上面。设置好断点之后，就可以令程序直接运行到该处，而不用再浪费时间去调试那些与问题无关的代码了。如果进入了一段与问题无关的代码，那么可以在其尾部设置断点，并且令程序继续向下执行，这样，程序就会在那个地方停下来，并把控制权交给调试器。

有些 bug，只有当程序沿着特定的路径执行时，才会表现出来，对于这种 bug 来说，我们应该把多个断点适当地结合起来使用。比如，有一个经常需要用到的例程叫做 c，它里面有一个 bug，只有当我们在测试用例 t 中调用该例程的时候，这个 bug 才能显现出来。如果直接在 c 上面设置断点，那么会浪费很多时间，因为只要有代码调用该例程，程序就会停下来，于是我们就必须反复执行 continue 命令，直到发现某一次调用是由测试用例 t 所发出的。比较好的做法，应该是先给 t 设置断点，等程序执行到这里之后，再给 c 设置断点（也可以提前给 c 设置断点，并将其禁用，等程序执行到 t 之后，再启用该断点）。

还有一类代码断点也很有用，它们令你能够在程序崩溃之前，有机会查看当前的状态。一般来说，当今的调试器都会在程序出现问题的时候，把执行过程打断，使得调试者可以在调试器里面进行某些操作。比如，如果程序遇到了未处理的异常，那么 Visual Studio 会弹出一个对话框，问你要不要在这里中断，而 gdb 工具则会在程序非正常退出的时候（例如，因为异常、信号或调用 abort() 而退出），打断其执行过程。如果使用的是 Eclipse，那么需要明确指出程序应该中断的时机。单击 Run-Add Java Exception Breakpoint 菜单，在弹出的对话框中搜索并指定相关的异常，如果程序碰到了这种异常，那么它的执行过程就会中断。

调试器的功能虽然很强大，但是它并没有超乎寻常的能力。如果程序在碰到某个反常的事件时，采用自己的代码来退出，那么程序就会直接停止，使得我们很难查看其在该事件发生时所处的状态。对于这样的情况来说，我们可以在程序处理这种事件所用的那部分代码上面添加断点。如果找不到那部分代码，那就把断点设置到经常用来退出程序的那个例程上面（如 exit、_exit 或 abort）。

如果还没有来得及添加断点，程序就已经挂起了（也就是停止响应了），那么可以通过调试器来强行停止这个程序（Eclipse ： Run-Suspend、Visual Studio ： Debug-Break

非卖品，仅供非商业用途或交流学习使用

All 或 Ctrl-Alt-Break、gdb：Ctrl-C）。程序停止之后，我们可以查看栈中的内容（参见第 32 条），以了解它刚才正在执行的是哪些语句。根据这些信息，一般就足以推测出使程序挂起的原因了。我们也可以从出现问题的那行代码开始，进行单步调试（参见第 29 条），以便更好地理解程序的行为。

有些场合下，我们并不关注某一行代码是在什么时候执行的，而是关注某一份数据是在什么时候修改的。这可以通过数据断点或观察点来实现。由于计算机每秒钟会执行数十亿条指令，因此想在其中把修改某个变量值的那条指令找出来，是极其困难的，尤其还要考虑到迷途指针[⊖]可能会改动内存中任意位置上面的数据。所幸当前的 CPU 都提供了适当的机制，使得调试器能够通过指定内存位置及对应的变量大小来设置数据断点。每次发生内存写入操作时，CPU 就检查正在写入的这个地址，是否位于指定的数据断点范围之内，如果确实位于该范围内，那么就打断程序的执行过程，并将其控制权交给调试器。由于 CPU 的这种检查机制与普通的内存写入操作是同时发生的，因此通常不会拖慢程序的执行速度。

由于数据断点与底层硬件贴得比较近，因此它设置起来不像代码断点那样固定。在 Visual Studio 中，可以通过 Debug-New Breakpoint-New Data Breakpoint 来设置这种断点。我们必须先把程序执行起来，使得全局变量的地址得以确定，然后才能设置这样的断点。设置断点时，需要指定的并不是有待观察的那个变量名，而是其内存地址（&variable）及数据尺寸（比如，整数一般是 4，双精度浮点数一般是 8）。使用 gdb 调试的时候，只需把变量名称作为参数，传给 watch 命令即可。（如果 gdb 没有给出一行写有 Hardware watchpoint 字样的回复信息，那就意味着你的工作环境无法从硬件层面支持观察点，此时 gdb 将以软件的方式来模拟，这会使程序速度慢好几个数量级。）对于 Visual Studio 和 gdb 来说，如果我们想在动态分配的变量（也就是通过 malloc 或 new 分配在堆上的变量，或是例程的局部变量）上面设置数据断点，那么必须先等这个变量确实存在（我们可以通过适当的代码断点来确保这一点），待其拥有一个已知的内存地址，然后才能进行设置。用 Eclipse 调试 Java 程序的时候，很容易就能设置观察点：只需打开对应字段的关联菜单，并选择 Toggle Breakpoint 即可。

调试器还能够设置带有条件的断点（必须满足一定的条件，才能够触发）、带有命中次数的断点（其出现次数必须达到一定的值，才能够触发），以及带有过滤器的断点

⊖ stray pointer（流浪指针），也称为 wild pointer（野指针）或 dangling pointer（悬挂指针）。——译者注

非卖品，仅供非商业用途或交流学习使用

（比如，必须位于特定的线程之内，才能够触发）。这些断点机制在某些情况下是很好用的，但如果你过份依赖它们，那就说明你应该考虑改用更为强大的调试手段了，例如，精准的测试用例（参见第 54 条）或丰富的日志记录（参见第 40 条）等。

要点

- 通过代码断点来缩减需要关注的代码范围。
- 如果某段代码会执行很多次，而其中只有少数几次是你所关心的，那么就先在其执行路径的上游设置断点，等程序暂停之后，再给这段代码设置断点。
- 如果要对非正常退出的情况进行调试，那么就针对异常或针对程序在退出时所调用的例程来设置断点。
- 如果程序失去了响应，那么可以在调试器里面令其停止执行。
- 用数据断点来锁定那些导致变量值意外改变的 bug。

第 31 条：了解反向调试功能

当前很多计算机都配有容量充足的内存和速度很快的 CPU，这通常意味着我们可以逆向执行程序。一般来说，为了实现逆向执行，调试器需要把每条指令对程序状态所做的修改记录下来，然后在逆向执行的时候，撤销这些修改。执行程序的过程中，你有可能发现自己错过了某一段可疑的代码，在这种情况下，反向调试的好处显得尤为突出。如果不做反向调试，那我们通常需要把断点设置得靠前一些，并且重新执行程序，然后单步进入那段可疑的代码中。问题在于：重新执行程序之后，可能要花费很长一段时间，才能够重现自己想要调试的那个 bug，而且如果在情急之下输错了命令，把正在调试的例程给跳出了，那么又得重来一遍。使用反向调试功能就不会这么麻烦了，发现自己错过了某一段可疑的代码之后，你可以逆向执行程序（等执行到一定的地方之后，或许需要转为正向执行），以便锁定刚才的那个 bug。

有很多独立的调试器与 IDE 都支持反向调试，只不过该功能的实现形式和名称有所不同。比如，Visual Studio IntelliTrace、Rogue Wave Software 的 TotalView ReplayEngine、Undo Software 的 UndoDB，以及 gdb 等，都支持反向调试。在这一类软件中，有的软件可以单独运行，有的软件是在 Eclipse CDT 平台下面运行的。由于反向调试还没有得到普及，因此各种软件对它的支持程度（以及各自的开销）会有所不同。

非卖品，仅供非商业用途或交流学习使用

下面我们用 gdb 把反向调试的过程简单演示一遍。假设有这样一个很短的程序需要调试。(为了缩减篇幅，笔者并没有在程序里面置入 bug，然而通过这个例子，你依然可以明白在实际的调试工作中，应该怎样退回到原来执行过的地方。)

```
 1  #include <stdio.h>
 2  int
 3  main()
 4  {
 5          int i, sum = 0;
 6
 7          for (i = 0; i < 10; i++)
 8                  sum += i;
 9          printf("%d\n", sum);
10  }
```

首先用调试器打开上述程序，并在 main 这个入口处设置断点。

```
(gdb) break main
Breakpoint 1 at 0x40050e: file loop.c, line 5.
(gdb) run
Starting program: /home/dds/a.out

Breakpoint 1, main () at loop.c:5
5           int i, sum = 0;
```

我们在 gdb 里面显示变量 i 及 sum 的值，然后在程序的最后一行代码上面设置断点。接下来，要执行一个关键的 record 命令，使得 gdb 能够把程序的执行经过记录下来，以便稍后进行反向执行。

```
(gdb) display i
(gdb) display sum
(gdb) break 10
Breakpoint 2 at 0x400542: file loop.c, line 10.
(gdb) record
```

等程度执行到刚才设置好的断点之后，我们发现，i 和 sum 的取值是正确的。

```
(gdb) cont
Continuing.
45
Breakpoint 2, main () at loop.c:10
10    }
2: sum = 45
1: i = 10
```

现在，可以用 gdb 的 reverse-next 命令来反向执行程序。

非卖品，仅供非商业用途或交流学习使用


```
(gdb) reverse-next
9           printf("%d\n", sum);
2: sum = 45
1: i = 10
(gdb) reverse-next
7           for (i = 0; i < 10; i++)
2: sum = 45
1: i = 9
(gdb) reverse-next
8                   sum += i;
2: sum = 36
1: i = 9
```

reverse-step 命令也用来进行反向执行。与正向执行时所用的 step 命令相似，该命令不会直接把某个例程执行完，而是会进入那个例程里面进行单步执行。reverse-continue 命令可以使程序一直反向执行下去，直至遇到断点。对于本例来说，这个断点就是 main 函数的入口，此时我们可以看到，那两个变量又回到了最初的值。

```
(gdb) reverse-continue
Continuing.

No more reverse-execution history.
main () at loop.c:5
5           int i, sum = 0;
2: sum = 0
1: i = 0
```

请注意，反向执行是有一些限制的。为了使程序能够反向执行，调试器必须在幕后做大量的工作，而反向执行所遇到的很多限制，正源自于此。于是，程序的性能可能会急剧下降、调试器所耗的内存可能会相当多，而且反向执行的距离也是有限度的（gdb 默认可以回退 20 0000 条指令）。此外，正在调试的这个程序，与其他系统及外部世界所做的某些交互操作，通常是无法撤销的，比如，已经打印到终端中的字符，依然会留在那里；已经从远程数据库中删除的数据行，不会重新出现；输入给程序的信号以及网络数据包等异步事件，同样很难在程序继续正向执行时重制。尽管如此，但是反向调试毕竟是个有用的功能，有时可以为你省下很多时间，使你不用再做一些枯燥而乏味的工作。

要点

- 了解反向调试功能。

第 32 条：查看例程之间的相互调用情况

程序在执行的时候，会开辟一种名为调用栈的数据结构，使得各个例程之间能够按照一定的顺序来相互调用。当某个例程调用另一个例程的时候，它要把下面这些数据推入栈中，或是分配在栈上面。（为了优化性能，有时可能会使用 CPU 寄存器。）

- 传给受调用的那个例程的参数。（它们通常按照从右至左的顺序入栈，以便使那种参数个数可变的例程，能够更加容易地对其进行处理。）
- 如果受调用的例程是一个方法，那么还要传入指向该方法所在对象的指针。
- 受调用的例程执行完毕后将要返回的地址。
- 受调用的例程所具备的局部变量。

程序可以根据上述元素在目前这个栈里面的相对位置来分别访问它们，这样做使得同一个例程能够多次得到调用（甚至是递归式的调用）。对于正在执行的程序来说，栈是极为重要的信息，其重要性仅次于程序当前的执行位置。

请注意，x86、x86-64 以及 68k（68000）等 CPU 架构，会通过调用约定来确保栈的实际使用方式与预期的方式相符。而 ARM、PowerPC 及 SPARC 等 CPU 架构，则会用 CPU 的寄存器来保存例程的参数，其中有些 CPU 还会把返回地址也放在寄存器中，对于这种用法来说，只有当 CPU 没有足够多的寄存器可以处理数据时，它才会把数据放在栈中。

每种调试器都能够查看栈信息（Eclipse：Window-Show View-Debug、Visual Studio：Debug-Windows-Call Stack 或 Alt-7、gdb ： where），程序为了到达当前这一点所调用过的每一个例程，都能与栈信息中的某一行对应起来。那一行里面写有与该例程有关的文件名或类名、例程本身的名称、例程的参数以及它上一次执行到的位置。这些信息，相当于一份描述程序状态的快照。程序当前执行的这个例程，会写在最上方，而程序的入口点（通常是一个名叫 main 的例程），则会出现在靠近底部的位置，有时它的下方，可能还写有一些引领程序到达这个入口点的其他例程（例如，Windows 程序的 wmainCRTStartup()），那些例程位于运行时程序库或内核中。

在栈跟踪信息中，有一些地方需要特别说明。

- 如果栈跟踪信息的顶部包含无法识别的例程，那说明你是在程序正执行第三方代码的过程中，将其打断的。对于 GUI 程序来说，如果它正在通过框架中的程序库来进行底层交互（如等待用户单击某个按钮），那么通常就会出现这种现象。

此外，如果程序在调用第三方库（例如，嵌入式的 SQL 程序库）来处理某些繁重的任务时被打断，那么也会出现这种现象。如果显示出来的例程不带参数，并且显示的是内存地址而不是例程名称，那就说明代码在编译的时候没有包含调试信息（参见第 28 条）。我们可以从上到下查看栈跟踪信息，以确定程序是从什么地方开始调用第三方代码的。

- 如果栈跟踪信息里面显示的全都是第三方例程，那说明程序可能是通过框架来运作的，那个框架在与需要和本程序进行交互的时候，会回调本程序中的相关例程，并把控制权交给程序。由于我们要调试的或许不是框架中的代码，而是自己程序中的代码，因此需要设法添加一些断点，使得程序能够在运行到这一部分代码时停下来。
- 如果栈跟踪信息的层次很浅，而且其中没有我们可以看得懂的例程，那可能意味着某个迷途指针把栈给破坏了。此时需要从早前的某个位置开始，仔细对代码进行单步调试，以确定调用栈是在什么地方遭到破坏的。
- 如果某个或某些例程在栈跟踪信息里面反复出现，那可能说明程序的递归代码有 bug。

根据调试器所显示出来的栈信息，我们很容易就能在例程之间游走。如果使用的是带有图形界面的调试器，那么只需要在栈信息中单击对应的行，就可以令程序进行上下文切换，从而跳转到与对应的栈帧有关的那个例程中。跳转过去之后，你可以看到那个例程当时执行到哪一行代码了，而且还可以查看它的局部变量与参数。如果使用的是 gdb 工具，那么可以通过 frame n 命令来把程序的上下文切换到第 n 帧，此外，也可以通过 up 和 down 命令，在栈帧之间上下移动。

要点

- 查看程序的栈信息，以了解其执行状态。
- 如果栈信息比较乱，那说明代码写得可能有问题。

第 33 条：查看变量及表达式的值，以寻找程序中的错误

如果想要迅速了解某个例程的关键变量，那么可以把局部变量的值展示出来（Eclipse：Window-Show View-Variables 或 Alt-Shift-Q V、Visual Studio：Debug-

Windows-Locals 或 Alt-4、gdb ： info locals 或 info args）。在写得比较好的代码中，局部变量的个数不会太多，而且仅凭这几个变量，就可以清楚地掌握程序的执行状况。如果表达式的意思太过复杂，很难通过局部变量的取值来理解，那么可以考虑引入一些名称较为合适的临时变量，以此来简化这个表达式。有人可能担心这样做会降低程序的效率，其实它对效率的影响是可以忽略不计的，因为即便有，编译器也会把它优化。

Gary Bernhardt 在 CodeMash 2012 上发表了一次题为〈 Wat 〉（What ？）的搞笑演讲，从他所进行的演示中，我们可以看到：表达式的真实取值，未必就是你所期望的那个值。因此，如果你想知道某段代码为什么运行得比较奇怪，那么不妨看看表达式的取值是否正确。如果这个表达式直接出现在代码中（而不是出现在 C/C++ 的宏里面），那么可以用鼠标选中它，然后单击关联菜单中的 Inspect（适用于 Eclipse），或是把鼠标悬停在表达式上面（适用于 Visual Studio）。你也可以查看任意表达式的取值：用 Eclipse 做开发的时候，可以单击 Window-Show View-Display 菜单，在窗口里面输入表达式，选中它，然后单击关联菜单中的 Inspect；用 Visual Studio 做开发时，可以通过 Debug-QuickWatch 菜单（或 Shift-F9 键）打开一个窗口，并在其中输入需要查看的表达式；用 gdb 调试程序的时候，可以用 print expression 命令打印表达式的值。请注意，只有当表达式所使用的变量在当前栈帧中具备定义的时候，调试器才能够评估这个表达式的取值。

我们通常还想在执行代码的过程中，观察一下表达式的取值会有什么样的变化。用 Eclipse 做开发时，可以通过 Window-Show View-Expression 打开一个窗口，并在其中输入想要观察的表达式；用 Visual Studio 做开发时，可以通过 Debug-Windows-Watch 打开类似的窗口；用 gdb 调试程序时，可以通过 display expression 命令来实现此功能。Visual Studio 会在表达式的取值发生改变时，以红色来凸现它，这使得我们在算法的执行过程中，能够更为容易地看出该表达式的变化情况。

有很多调试器扩展机制、程序库以及工具，都可以帮助你查看并理解复杂的数据类型与数据结构，例如，任意精度的数字、树（tree）、图（map）以及链表（linked list）等。

- 在 Visual Studio 中，可以编写 custom visualizer，以便用适当的格式来显示某个对象。
- 在 gdb 中，对应的机制叫做 pretty-printer。
- 在 QtCreator 中，对应的机制叫做 debug visualizer。
- 用 Python 编程时，可以引入 pprint 模块，并通过其 PrettyPrinter 方法来打印数据。
- 用 Perl 编程时，可以使用 Data::Dumper 模块来打印数据。
- 在 JavaScript 代码中，可以用 JSON.stringify(obj, null, 4) 来显示对象的内容。

- 我们可以把 Python、Java、JavaScript 或 Ruby 代码粘贴到 Python Tutor 软件的窗口里面，对其进行单步执行，并查看对象与栈帧之间的相互指向关系（参见图 4.1）。
- 如果其他办法都不合适，那么还可以编写一个简短的数据转换脚本，将其输入到 Graphviz 绘图程序里面。

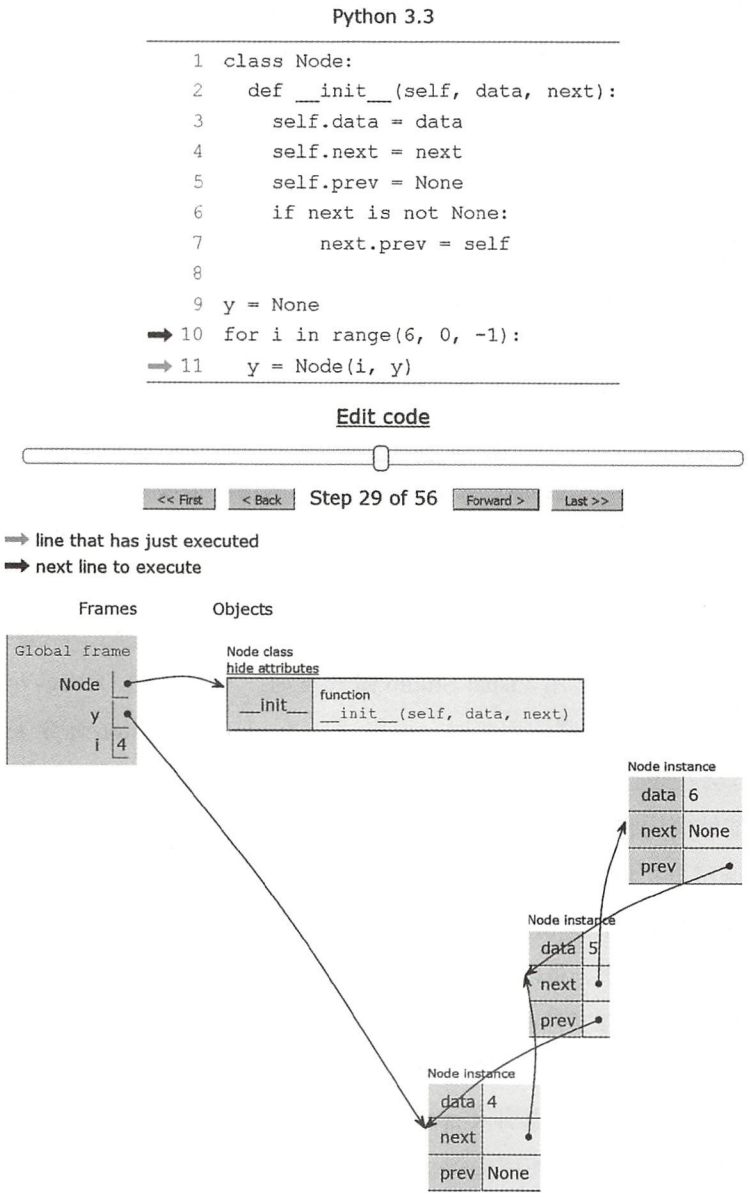


图 4.1　在 Python Tutor 软件中，以图形化的方式来查看双向链表程序的执行情况

要点

- 对重要的表达式进行验证，看看它们的值是否正确。
- 对调试器进行设置，令其能够在算法的执行过程中，持续地显示出表达式的变化情况。
- 通过局部变量来了解例程的运行逻辑。
- 用数据可视化机制来展示复杂的数据结构。

第 34 条：了解怎样把调试器连接到正在运行的进程上

有时我们通过大量的努力，或是在偶然的机缘之下，重现了一个相当重要而又难于捕获的 bug。可是问题在于，表现出该 bug 的这款应用程序，并不是从调试器里面运行的，如果现在又要从调试器里面启动一次，那么或许就很难重现它了。面对这样的问题，我们应该怎么办呢？

所幸调试器能够把自己连接到正在运行的线程上面。如果要用 Visual Studio 来调试，那就通过 Debug-Attach to Process 菜单来连接，如果要用 gdb 来调试，那就先找到待调试的那个程序所对应的 ID 号。运行 ps 命令，并在输出结果中查看 PID 栏，即可找到这一信息。ps 命令的参数及输出的样式，在各系统上面有所不同。下面这三条命令，可以分别在 GNU/Linux、OS X 及 FreeBSD 系统上面根据给定的用户名来查询相应的 Web 服务器进程。

```
$ ps -u apache
  PID TTY      TIME CMD
25582 ?    00:00:00 httpd
25583 ?    00:00:00 httpd
$ ps -u www
  UID  PID TTY       TIME CMD
   70  299 ??     0:00.02 /usr/sbin/httpd -D FOREGROUND
   70 5363 ??     0:00.00 /usr/sbin/httpd -D FOREGROUND
$ ps -U www
 PID TT STAT  TIME COMMAND
1045 - S   33:39.19 nginx: worker process (nginx)
```

然后，以带有进程 id 的 -p 选项来运行 gdb 命令，以便与待调试的程序相连接。

```
$ sudo gdb -p 25582
(gdb) where
```

```
#0 0x00007f75e115d017 in accept4 () from /lib64/libc.so.6
#1 0x00007f75e18687ba in apr_socket_accept () from /usr/lib64/
   libapr-1.so.0
#2 0x000055f2aff25513 in ap_unixd_accept ()
#3 0x00007f75d859f717 in ?? () from /etc/httpd/modules/
   mod_mpm_prefork.so
#4 0x00007f75d859f9d5 in ?? () from /etc/httpd/modules/
   mod_mpm_prefork.so
#5 0x00007f75d859fa36 in ?? () from /etc/httpd/modules/
   mod_mpm_prefork.so
#6 0x00007f75d85a0710 in ?? () from /etc/httpd/modules/
   mod_mpm_prefork.so
#7 0x000055f2afef000e in ap_run_mpm ()
#8 0x000055f2afee97b6 in main ()
```

如果想把调试器连接到 Java 程序，那么在用 JVM 启动那个 Java 程序时，必须传入下面这样的参数：

```
-agentlib:jdwp=transport=dt_socket,address=127.0.0.1:8000,\
server=y,suspend=n
```

如果用上面的参数来运行 Java 程序，那么该程序就可以在同一台计算机（可以用 127.0.0.1 这个 IP 地址来表示，也可以用 localhost 来表示）的 8000 端口上面，监听从调试器方面传来的 TCP 连接。如果要运行很多个有待调试的 Java 进程，那么必须给每个进程指定互不相同的端口，我们可以在 1024 至 6 5535 这个范围内，选择某个尚未占用的端口。netstat-an 命令能够列出处于 listen 状态的 TCP 连接，由此可以看到当前有哪些端口已经占用了。如果要用 Eclipse 来调试程序，那么接下来就单击 Run-Debug Configurations-New launch configuration，并按照之前启动 Java 程序时所指定的参数，填写 Host 和 Port 字段，然后单击 Debug，将调试器连接到指定的进程。此外，如果发现某个 Java 进程运行得不太正常，那么可以把该进程的 id 传给 jstack 命令，这会显示出正在运行的每一条线程所对应的栈转储信息，我们由此就能够知道该进程到底在做些什么。

把调试器连接到正在运行的进程之后，可以打断其执行过程（Eclipse:Run-Suspend、Visual Studio:Debug-Break All、gdb:break），并查看其调用栈来检视该进程当前究竟在什么样的情境中执行（参见第 32 条）。此外，也可以设置一些断点，使程序在运行到这些断点的时候得以中断。在程序的运行过程中，还可以查看那些在整个程序范围内都可以访问的变量及对象。

有时可能要调试一个运行在其他主机上面的程序。之所以要这样做，可能因为我们

非卖品，仅供非商业用途或交流学习使用

使用的是带有图形界面的调试器，而执行程序的那台计算机却没有图形界面，或者没有提供远程访问其图形界面的方式，这种情况，通常出现在执行嵌入式软件或代码的服务器上面。还有一种情况，是因为待调试的程序运行在缺乏相关资源或底层设施的设备上面，这种设备没有办法开启功能完备的调试器。面对这两类情况，我们需要在远端运行一款小型的调试监视器，并且把自己的调试器与之相连。具体细节会根据调试器及连接类型而有所不同，请您查阅与自己的工作环境有关的文档。

下面来演示如何用 gdb 进行远程调试。假设要对电视上面的某一款录像程序进行调试，那就先在电视端启动调试服务器，令其监听（未占用的）1234 端口，使得工作计算机中的客户端能够通过该端口与调试服务器建立 TCP 连接。

```
$ gdbserver your-workstation.example.com:1234 video-recorder
```

然后，在自己的工作计算机上面，以相同的可执行文件为参数来运行 gdb 命令。接下来，通过 target 命令指定远程主机和端口。

```
$ gdb video-recorder
(gdb) target remote tv-12.example.com:1234
Remote debugging using tv-12.example.com:1234
```

现在就可以在自己的工作计算机上面，像执行常规的 gdb 命令那样来控制并查看电视中的 video-recorder 进程了。如果你认为这种远程调试的办法太过复杂，那么可以考虑给远程主机安装一款带有图形界面的调试器，并将其窗口展示到自己的计算机中，或是把它的屏幕映射到自己的显示器上面（参见第 18 条）。

要点

- 把调试器连接到正在运行的进程上面，以便对其进行调试。
- 通过远程调试机制，对运行在资源受限设备上面的应用程序进行调试。

第 35 条：了解怎样运用核心转储信息来进行调试

我们通常可以在应用程序崩溃或终止之后对其进行调试。运行在 Unix 系统上面的原生应用程序能够生成核心转储文件，这是系统对应用程序崩溃时的情况所做的内存镜像。（由于刚刚引入这个概念时，业界还在使用磁芯内存，因此这种文件就称作核心转储。）要想生成核心转储文件，我们需要满足操作系统所提出的相关要求，并做出相

非卖品，仅供非商业用途或交流学习使用

应的设置。有的情况比较简单，应用程序只需要能够在其当前目录里面，创建名为 core 的文件即可，有的情况则复杂一些，例如，可以在 Linux 系统上面进行配置，把核心转储信息通过管道发给指定的程序。最重要的一个问题是：用户所能生成的核心转储文件，是有容量限制的，这个限额通常设置为 0。我们一般可以通过 ulimit-c 命令来查看限额，并且可以在该命令后面补充一个参数，以 KB 为单位来指定核心转储文件的最大容量。如果想要把这个修改后的值固定下来，那么可以把该命令添加到某位用户或整个系统的 shell 配置文件中，例如，可以添加到 .bash_profile 或 /etc/profile 里面。

如果想用 gdb 来查看 Unix 核心转储镜像，那么就把程序名称与核心转储文件的名称，作为参数传给 gdb。然后，可以用 where 命令查看程序在发生崩溃时所执行到的位置，也可以在调用栈里面的各个栈帧之间游走，还可以对表达式进行求值，以判断程序的状态。由于核心转储文件特别有用，因此你可能想要在程序遭遇内部错误时，生成这样的一份文件。这可以通过调用 abort 函数来实现。

在 Windows 系统中，我们要用另外一套流程来对原生应用程序进行事后调试。由于 Windows 程序默认并不会生成转储文件，因此必须在应用程序里面调用 MiniDumpWriteDump 函数。与 abort 函数类似，这个函数也应该是要等到程序出现内部错误的时候，才去调用的。此外，当程序发生异常（比如，访问了非法的内存地址）时，我们可能也希望 MiniDumpWriteDump 能够得到调用，这可以通过 SetUnhandledExceptionFilter 函数来实现：把调用 MiniDumpWriteDump 的那个函数，作为参数传给 SetUnhandledExceptionFilter 即可。有了转储文件之后，可以用 Visual Studio 打开该文件，以查看应用程序在生成转储时所处的状态。

有时应用程序虽然没有崩溃，但是却失去了响应，或进入了一种你需要对其进行调试的状态，此时你可能并不想把调试器连接到这个程序上面，而是想创建一份内存转储文件，以便把程序状态永久保存下来，将其发给同事，或是留着稍后进行调试。在 Unix 系统中，可以通过 kill-ABRT process-id 命令给程序发送 SIGABRT 信号，从而迫使其生成转储文件。（查询进程 id（process-id）的办法，请参见第 34 条。）如果程序运行在终端窗口里面，那么也可以通过按下 Ctrl-\ 组合键来发送这个信号。在 Windows 系统中，可以打开 task manager（任务管理器），右击相关的进程，然后选择 Create Dump File 菜单项，这样就会在临时目录中生成一份转储文件。（这个目录指的是 TEMP 环境变量所对应的目录，可以运行 %TEMP% 来查看其内容。）

如果你使用的是一种运行在托管环境中的语言，如 C#、Java、JavaScript、Perl、

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Python 或 Ruby 等，那么它们对事后调试的支持情况可能会各有不同，或是要根据具体的厂商来定，有些语言甚至根本就不提供事后调试机制。之所以会这样，是因为它们所使用的底层技术更加抽象、更加复杂，并且会随着实现情况而有所变化（你可以想象一下即时编译的实现方式），于是，操作系统就没有办法直接创建内存转储文件了，而且有的时候（比如，有上千个线程正在执行，或是有很多 JavaScript 事件有待处理），创建这种文件会对计算机提出很高的要求。根据笔者编写本书时的情形来看，很多托管环境要么不提供事后调试机制，要么就仅仅提供一些很基础的支持。如果你是在这样的环境之下做开发的，那么在了解到与事后调试有关的知识之后，可以查阅当前的文档，看看该环境能够提供何种程度的支持。

事后调试机制的常见用法，是打造一套流程，使我们可以根据该流程，从客户那里获取内存转储文件，并对其进行调试。这样的一套流程，应该做到以下几点。

1. 必须使自己所写的程序能够创建出内存转储文件，以及与之相关的元数据（metadata）。内存转储文件的创建方式，刚才我们已经讲过了，而至于元数据，则至少应该包含程序的版本，此外，还可以包含与程序的执行环境有关的数据（如处理器、操作系统、环境变量、共享库的版本）、日志文件、客户所输入的数据（要把客户的机密保护好）以及程序的历次使用情况（这部分内容可以存放在日志文件里面）。

2. 程序生成了内存转储文件之后，必须要有适当的办法将其发送出去。如果程序崩溃之后，其状态已经遭到了破坏，那么最好是通过另外一个程序来发送这份文件。我们可以用那个程序来运行本程序，并检查本程序的退出状态，只要发现这个状态码和预先达成的约定值相符，那就把内存转储文件发送出去。这个办法要想奏效，我们必须在 SIGABRT 信号处理程序（适用于 Unix 系统）或异常过滤器函数（适用于 Windows 系统）中，把该值用作本程序的退出值。内存转储数据可以通过 HTTP POST 请求来发送，这或许是最为简单而可靠的做法。

3. 在你工作的地方编写一个小型的服务器程序，用来接收相应的 HTTP 请求，并把收到的数据保存起来，以便进行深入的分析。其中的某些元数据或许应该放在数据库里面，这样分析起来会更容易。

4. 设计出一种办法，以便对发生崩溃的这个版本进行调试。从源代码这一方面来说，只要保证自己在版本控制系统里面，给每个发行版所对应的源代码都贴过标签（参见第 26 条），并确保客户所发来的元数据里面也含有这种标签（tag）就可以了。而从可执行文件这一方面来说，由于很难保证自己可以根据某个版本的源代码来重建一个与

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

之前完全相同的程序（需要考虑到其中嵌入的时间戳、代码的随机化以及编译器的更新等因素），因此，最好是在提交每个版本的源代码时，把编译出来的二进制文件也一起提交到版本控制系统里面（如果是 Windows 程序，那么还要把相关的 PDB 调试文件放进去）。

5. 能够使用正确的源代码、可执行文件以及与之相应的内存转储文件来启动调试器，以便对发生崩溃的这个版本进行调试。

如果你觉得这套流程构建起来比较复杂，那么可以采用框架完成这些任务，比如，可以考虑采用 PLCrashReporter 框架（适用于 iOS 及 OS X），并在该框架上面搭建服务，或使用由 Crittercism、New Relic 或 Splunk MINT 所提供的托管服务。

要点

- 获取并检视程序的内存转储信息，以便对发生崩溃和失去响应的应用程序进行调试。
- 打造一套错误报告系统，以便对客户所安装的应用程序进行调试。

第 36 条：把调试工具设置好

对于调试工作来说，我们应该用心营造一套基本的工作环境（参见第 9 条），这其中就包括对调试器进行设置。下面给出一些思路。

首先，要使用图形化的用户界面。尽管笔者喜欢命令行界面，而且觉得自己在用这种界面工作时，效率会变得很高，但是毕竟有少数几种工作，更适合用 GUI 界面来完成，调试就是其中之一。由于调试的时候要同时查看源代码、局部变量、调用栈、日志消息等多种数据，因此图形化的界面更有优势。如果你是用 Eclipse 或 Visual Studio 来调试程序的，那就不用考虑 GUI 的问题了，因为这些 IDE 本身就具备图形界面。如果你是用 gdb 来调试程序的，那么可以考虑改用其他一些工具。功能上较为完备的替代产品是 DDD(DataDisplayDebugger)，它是个基于 Unix 的前端工具，含有完备的图形界面，不仅可以支持 gdb，而且还能支持其他一些运行在命令行界面中的调试器，例如，那些适用于 Perl 语言、Bash shell(bashdb)、make(remake) 以及 Python 语言（pydb）的调试器。此外，它还可以很好地展示程序的数据结构。如果你的程序运行在 Unix 主机上面，那就用 DDD 来做调试吧。（只要客户端计算机的操作系统支持 X server 就可以，Windows

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

与 OS X 都属于这样的系统。）还有一个办法是用 -tui 选项来运行 gdb，或是在运行了 gdb 之后，输入 - 命令，进入基于文本的用户界面（text-based user interface，tui）。如果你不习惯 Emacs 风格的快捷键，那么可以考虑改用 cgdb，它的界面与 vi 比较接近。

还有一种办法也可以令你更加流畅地使用调试器，那就是把调试时所用到的一些实用命令保存到文件里面，以后每次开始调试时，都把这份文件执行一遍。Visual Studio 和 Eclipse 会自动把调试过程中所进行的设置（例如，你所添加的断点与观察点），保存到当前的项目中。gdb 虽然设置起来稍微有点困难，但却可以做得更加灵活一些。我们可以把 gdb 命令放在名为 .gdbinit 的文件里面，并将其保存到 home 目录或项目所在的目录中，前者可以在每次启动 gdb 时得到执行，后者可以在每次进入项目的目录并运行 gdb 时得到执行。此外，也可以把 gdb 命令放在其他文件里面，然后在启动 gdb 时，通过 -x 选项加载该文件。这三种办法还可以合起来用：

- 把每次启动 gdb 时都要执行的那些命令，放在 $HOME/.gdbinit 里面。
- 把针对当前项目而定义的那些内容，放在 myproject/.gdbinit 里面。
- 把有助于调试 1234 号事务的那些断点和观察点，放在名为 issue-1234 文件里面。

在全局的 .gdbinit 文件中，可以放入一个特别有用的命令，这就是 set history save，它可以令 gdb 把你每次在调试时所输入命令保存起来，下次启动 gdb 之后，你依然可以通过键盘的上下箭头或通过搜索来找到那些命令。此外，你可以在 home 目录里面放入名为 .inputrc 的文件，并在其中编写一些命令，使得 gdb 的输入界面所使用的快捷键，能够符合你的习惯。比如，set editing-mode vi 或 set editing-mode emacs 命令，可以分别把快捷键设置得与 Emacs 或 vi 编辑器所使用的那套快捷键相符。

全局的 .gdbinit 文件，还适合用来给那些常用的命令定义别名。比如，下面这个脚本定义了名为 sf 的新命令，用来显示当前的栈帧。

```
define sf
  where
  info args
  info locals
end
document sf
Display current stack frame
end
```

GitHub 网站上有一个 Gist 叫做 gdbinit，它给出了很多这样的用法。

我们可以用更为复杂的脚本来调试一些困难的问题，当我们不想或不能在程序的源代码中添加断言的时候（参见第 43 条），这么做尤其有用。比如，如果程序所使用的资

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

源必须用一对 lock-unlock 块括起来，而且最多只应该用一个这样的块括起来，那么就可以先运行程序清单 4.1 中的 gdb 脚本，然后再执行程序，看看它在资源的保护上面，有没有遵守刚才所说的那条规则。

程序清单 4.1 用来对锁定顺序进行检查的 gdb 脚本

```
# Define a counter variable to keep track of locks
set $nlock = 0

# Stop the execution with a backtrace on nested locks
break lock if $nlock > 0
commands
  silent
  echo Nested lock\n
  # Display the stack trace
  backtrace
  # Stop the program's execution
  break
end

# Stop the execution with a backtrace on duplicate unlocks
break unlock if $nlock <= 0
commands
  silent
  echo Duplicate unlock\n
  backtrace
  break
end

# When the lock routine is called, increase the counter
# Define a new breakpoint
break lock
# Commands to execute when the lock routine is called
commands
  silent
  # Increment counter variable
  set $nlock = $nlock + 1
  # Continue the program's execution
  continue
end

# When the unlock routine is called, decrease the counter
break unlock
commands
  silent
  set $nlock = $nlock - 1
  continue
end
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

上面这段脚本，首先定义了名为 $nlock 的计数器变量，以统计当前的锁定深度。接下来，它给 lock 和 unlock 这两个函数，分别设置有条件的断点，用以检查 $nlock 变量的值是否合理，如果发现这两个函数的调用顺序不对，那就中断执行过程并打印出错误消息。此外还有两个断点也分别设置在这两个函数上面，用以维护 $nlock 变量的值。请注意，这两个有条件的断点，必须定义在那两个无条件的断点之前。

如果 gdb 本身的功能无法满足你的需求，那么可以考虑采用新版 gdb 所提供的 compile 命令来编译并运行代码。这些代码使用与应用程序相同的原生语言来撰写，并且能够访问局部和全局的变量及函数。如果我们用这种编译后的元素来定义新的调试命令，那么就可以借此实现出很多功能，但是要注意：复杂一些的调试机制，通常还是应该实现在程序自身的源代码里面，以便于对其进行维护、分享及归档（参见第 40 条）。除非确实有强烈的理由使得我们无法这么做，否则，就不应该把所有的基本调试机制都用 gdb 命令来实现。

有时在修改了程序的源码之后，为了保留本次调试过程中所输入的一些命令，我们并不想重新启动 gdb。有一种方式能够做到这一点，那就是在 gdb（或另外一个窗口）里面运行 make 命令。当我们重新运行程序时，gdb 会判断出程序的可执行文件已经发生了变化，因而会自动把它重新加载一遍。

要点

- 使用带有图形界面的调试器。
- 对 gdb 进行配置，使它能够把输入过的命令保存下来，并设置一套符合自己使用习惯的快捷键。
- 把常用的命令放在 gdb 脚本中。
- 修改完源代码之后，可以不重新启动 gdb，而是直接在 gdb 里面构建程序，以便保留你在这次调试会话中所输入过的命令。

第 37 条：学会查看汇编代码及原始内存

我们在调试程序时或许会发现：即便是很简单的一行代码，也依然有可能无法像预期的那样运作。要想找到问题的原因，我们可以去看看程序的底层代码究竟是如何执行的，在这个层面，我们所看到的内容就是程序真正执行的内容，也就是说，我们所

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

看到的每一条机器指令，都对应一项简单的操作，而不会包含一些夹杂着隐晦问题的抽象层。查看这些正在执行的机器码，可以帮助我们发现各种各样的错误，例如，执行了不必要的类型转换、误解了操作符的优先级规则、无意中使用了重载之后的操作符、使用了没有配对的括号、使用了错误的数值类型（如在本来应该用 long 的地方使用了 int）或是使用了不当的多态例程等。（其中某些问题可以通过单步调试来进行深入探查，但是代码内联（code inlining）机制或许会使你无法这样做。）

机器指令解读起来，其实并没有想象的那样困难。比如，程序清单 4.2 中的这个 C 语言程序，可以采用两种不同的格式来编译（未加优化的）汇编代码，一种格式是程序清单 4.3 这样的 ARM 代码，它采用 AT&T 汇编语法（assembler syntax），这种格式经常出现在 Unix 系统上面，另一种格式是程序清单 4.4 这样的 Intel x86 代码，它采用 Intel 汇编语法，这种格式经常出现在 Windows 系统上面。大多数汇编指令都是顾名思义的，例如，add（加法）、mov（move，移动）、cmp（compare，比较）以及 call（调用）等。与高级语言中的变量不同，汇编语言采用少数几个固定的寄存器来保存数据，例如，Intel 处理器的 eax 或 edx 寄存器，以及 ARM 处理器的 r0 至 r15 等寄存器。在大多数情况下，其中的某一个寄存器（eax 或 r0）会用来表示函数的返回值。写在方括号里的值表示内存地址，这种写法用来访问该地址中的内容。局部变量与例程参数，是根据其与栈中某个内存地址之间的偏移量来访问的，这个参照地址，称为 frame pointer（帧指针）寄存器（也就是 ebp 寄存器或 fp 寄存器）。我们可以通过 push 及 pop 指令，或是通过修改帧指针的值来对栈进行操作。循环是通过跳转指令来实现的，也就是说，其尾部会出现 jmp 或 b（代表 branch，转入分支）指令，以便跳转到开头。如果想实现有条件的操作，那就先用 cmp 指令来比较两个值，然后根据比较的结果来执行条件跳转指令（例如，jle 指令会在小于或等于的情况下进行跳转、ble 指令会在小于或等于的情况下转入分支）。浮点操作采用另外一套寄存器和与之对应的一套指令来实现。

程序清单 4.2　一个简单的 C 语言循环计数程序

```
#include <stdio.h>
main()
{
        int i;
        for (i = 0; i < 10; i++)
                printf("%d\n", i);
        return 0;
}
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

程序清单 4.3　用 AT&T 语法对上述 C 语言程序进行编译所得到的 ARM 汇编代码

```
.section    .rodata     @ Data area
.align 2                @ Align on even memory address
.LC0: .ascii "%d\012\000"     @ Printf format string
      .text                   @ Code area
      .global main            @ Export main
main:                         @ Entry point of main
      stmfd sp!, {fp, lr}     @ Function entry boilerplate
      add   fp, sp, #4
      sub   sp, sp, #8
      mov   r3, #0            @ Set register r3 to zero
      str   r3, [fp, #-8]     @ Store register r3 into i
      b     .L2               @ Branch to loop's end
.L3:                          @ Loop's top label
      ldr   r3, .L4           @ Get printf format address
      mov   r0, r3            @ Set format as first argument
      ldr   r1, [fp, #-8]     @ Set i as second argument
      bl    printf            @ Call printf
      ldr   r3, [fp, #-8]     @ Get i into register r3
      add   r3, r3, #1        @ Increment r3 by one
      str   r3, [fp, #-8]     @ Store register r3 into i
.L2:                          @ Loop's end label
      ldr   r3, [fp, #-8]     @ Get i into register r3
      cmp   r3, #9            @ Compare r3 with 9
      ble   .L3               @ If less or equal then
                              @ branch to loops top
      mov   r3, #0            @ Set r3 to zero
      mov   r0, r3            @ Zero r0 as main's return value
      sub   sp, fp, #4        @ Function exit boilerplate
      ldmfd sp!, {fp, pc}
.L4:  .word .LC0              @ Address of printf format arg
```

程序清单 4.4　用 Intel 语法对上述 C 语言程序进行编译所得到的 x86 汇编代码

```
_DATA SEGMENT                 ; Data area
$SG2748 DB  '%d', 0aH, 00H    ; printf format string
_DATA ENDS

PUBLIC _main                  ; Export main
EXTRN _printf:PROC            ; Import printf

_TEXT SEGMENT                 ; Code area
_i$ = -4                      ; Stack offset where i is stored
_main PROC
      push  ebp               ; Function entry boilerplate
      mov   ebp, esp
      push  ecx
      mov   DWORD PTR _i$[ebp], 0 ; i = 0
      jmp   SHORT $LN3@main   ; Jump to loop's end
```

非卖品，仅供非商业用途或交流学习使用


```
$LN2@main:                      ; Loop's top label
      mov   eax, DWORD PTR _i$[ebp] ; Get i into register eax
      add   eax, 1              ; Increment by one
      mov   DWORD PTR _i$[ebp], eax ; Store eax back to i
$LN3@main:                      ; Loop's end label
      cmp   DWORD PTR _i$[ebp], 10  ; Compare i to 10
      jge   SHORT $LN1@main     ; If greater or equal
                                ; terminate loop
      mov   ecx, DWORD PTR _i$[ebp] ; Get i into register ecx
      push  ecx                 ; Push ecx as argument to printf
      push  OFFSET $SG2748      ; Push printf format string
                                ; argument
      call  _printf             ; Call printf
      add   esp, 8              ; Free pushed printf arguments
      jmp   SHORT $LN2@main     ; Jump to loop's top
$LN1@main:                      ; Loop's exit label
      xor   eax, eax            ; Zero eax as main's return value
      mov   esp, ebp            ; Function exit boilerplate
      pop   ebp
      ret   0
_main ENDP
_TEXT ENDS
END
```

用 Visual Studio 做开发时，可以通过 Debug-Windows-Disassembly 菜单或 Alt-8 键打开一个窗口，以查看反汇编之后的代码，并使用与 step-into 及 step-over 相对应的快捷键来进行单步调试，还可以通过 Debug-Windows-Registers 菜单或 Alt-5 键查看寄存器的值。用 gdb 调试程序时，可以执行 display/i$pc 命令，以便在每一步的调试过程中显示出反汇编之后的指令，然后，可以用 stepi 及 nexti 命令来单步调试。你可以通过 info registers 命令查看寄存器的值，也可以通过 display$r0 或 display$eax 这样的命令来持续地显示某个具体的寄存器。如果用 Eclipse 做开发，那么可以安装 bytecode 插件，以显示反汇编后的 JVM 字节码。

通过查看寄存器的值，也可以了解函数中的 return 语句所计算出来的返回值。只需要在函数即将返回到其调用方的时候，把相关寄存器的值显示出来就可以了。请注意，如果函数要返回的对象比寄存器大，那么它通常会把该对象返回到栈中。

熟悉计算机的内部表示形式，也可以帮助你更好地进行底层调试工作。如果程序会从磁盘或其他进程里面读取二进制数据，那么我们可以检视对应的内存，以查看这些数据的内容。用 Visual Studio 做开发时，可以通过 Debug-Windows-Memory 菜单或 Alt-6 键来开启一个显示内存数据的窗口，然后在其地址栏中，输入你要查看的那个缓冲区数

非卖品，仅供非商业用途或交流学习使用

组的名称，或是输入一个像 &structure_variable 这样，可以产生内存地址的表达式。你也可以在显示内存数据的区域右击鼠标，并指定待查数据的尺寸及类型（例如，4 字节的有符号整数）。用 Eclipse 做开发时，可以通过 Window-Show View-Other-Debug-Memory 菜单来打开与之类似的 memory monitors 功能，然而这个功能似乎并不适用于 Java 程序。用 gdb 调试程序时，可以用 x/ 形式的命令来查看内存，我们在 x/ 后面写上需要查看的元素数量、与打印格式相对应的字符、每个元素的大小以及内存区块的地址。比如，x/10xb&a 命令会以字节（命令中的 b 表示字节）为单位，用十六进制（命令中的第二个 x 表示十六进制）显示出 a 中的 10 个元素。

查看数据在内存中的表示形式时，有一个地方需要注意，那就是：同一个整数可以用两种顺序来保存。第一种叫做小端序（little-endian，小端在前）格式，也就是先保存最低有效字节（least significant byte），然后依次保存权重较高的字节。Intel 架构以及大多数 ARM CPU，都采用这种格式。如果某个整数变量的值是 0x76543210，那么它在内存中就会表示成：

```
0x10  0x32  0x54  0x76
```

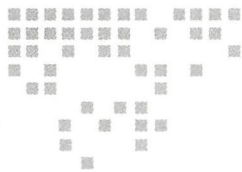
第二种格式叫做大端序（big-endian，大端在前）格式，也称为网络序格式。只有 SPARC 及 PowerPC 等较为少见的 CPU 架构才会采用这种格式。然而该格式依然是值得注意的，因为 TCP/IP 等重要的 Internet 协议，会采用这种格式来表示数据，而且 Java 在读取和写入二进制数据时，也会采用这种格式来进行操作。同样一个整数变量，如果改用这种格式来表示，那么它在内存中就会写为：

```
0x76  0x54  0x32  0x10
```

要点

- 查看反汇编后的机器指令，以了解程序代码的底层运作方式。
- 查看 eax 或 r0 寄存器，以了解函数的返回值。
- 查看数据在内存中的表示形式，以了解其在底层的存储方式。

非卖品，仅供非商业用途或交流学习使用



Chapter 3 第 5 章

编程技术

开发者所遇到的大多数故障，都与软件代码有关。要想找到与这些故障相对应的程序缺陷，其中一个办法，就是在某些机制的帮助之下仔细审视程序的代码。

第 38 条：对可疑的代码进行评审，并手工演练这些代码

为了锁定算法中的 bug，我们通常会仔细审视这些代码，或用手算的办法来推演其执行效果，以验证其编写方式以及自己对这段代码的理解是否正确。如果在审视代码的过程中找到了错误，那么这个任务就算完成了，若是找不到，则可以用调试器来执行这段代码（参见第 29 条），看看它在计算机上的实际运行效果与你所认为的效果之间有什么区别。

在第一轮的查看过程中，要仔细观察每一行代码，看看其中有没有常见的错误。其实，开发者只需要遵循适当的编程约定（例如，通过添加括号来厘清运算符之间的优先级），就可以避免很多常见的错误，即便出了错，也可以通过静态分析工具把它们找出来（参见第 51 条）。然而，还是有一些错误会漏过去，尤其是那些没有遵守编程约定的代码。需要关注的错误包括：操作符的优先级有误（位操作需要特别注意）、缺少必要的括号和 break 语句、（在控制语句后面）多写了分号、把比较操作误写为赋值操作、没有对变量进行初始化或将其初始化为错误的值、循环中缺少必要的语句、比正确的结果多 1 或少 1（off-by-one error）、类型转换错误、缺少必要的方法、拼写错误，以及与特定

编程语言有关的陷阱。

如果要手工执行代码，那就准备一张白纸，在上面写出关键变量的名称，然后按照计算机运行这段程序时所采用的顺序来执行这些语句（参见图 5.1）。变量的值如果发生了变化，那就划掉旧值并写上新值。为了修改起来方便，我们应该用铅笔写这些值。

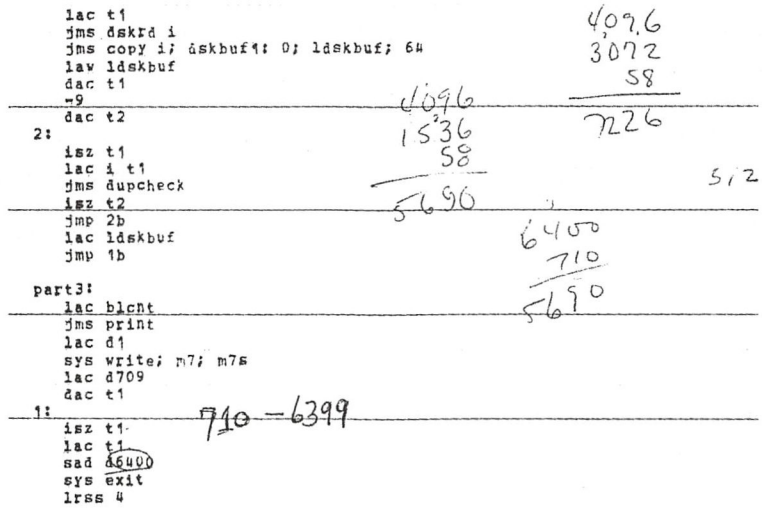


图 5.1 在一份写于 1970 年的代码旁边所进行的手工演算，这份代码是 check 程序的一部分，用来检查 PDP-7 的 Unix 文件系统，它是 fsck 程序的先驱

可以拿一个计算器过来（是真的计算器，不是计算机里的计算器程序），帮你迅速求解那些复杂的表达式。如果要做位运算，那么可以考虑使用程序员专用的计算器。不要用计算机软件来辅助这个手算的过程，因为该过程最需要的就是专注，如果你总是想把变量往电子表格里面写，总是想用编辑器来浏览代码，总是想去检查有没有新的电子邮件，那么你的专注程度就会下降。

如果代码会操作一些复杂的数据结构，那就用线条、方框、圆圈和箭头来表示这些情况。你需要构想一套图示，以便将算法中最重要的那些部分画出来。比如，在表示区间的时候（很多算法都会在区间上面犯错），可以用方括号形式的线条来表示区间中闭合的那一端（很多程序所用的区间，其起始端都是闭合的），用圆括号形式的线条来表示区间中开放的那一端[⊖]。把程序中你所关注的那一部分调用关系（也就是那一部分例程之间的相互调用情况）画出来，通常也是很有用的。如果熟悉 UML，那就按照 UML 规范

⊖ 闭合端（closed end）表示该区间包含端点值，开放端（open end）表示该区间不包含端点值。——译者注

来画图，但是请别在图示上面耗费过多的时间，你需要在绘图的方便程度与它的表意能力之间求得平衡。

拿一张比较大的纸来画，这样可以画出更多的内容。也可以考虑画在白板上面，它的空间要比一般的纸更大，而且便于擦除，也有利于多人讨论。用不同的颜色来区分图中的各类元素。如果你画的这张图很重要，那就把它拍下来，并关联到对应的事务上面。

还有一种更高级的玩法，是在绘图的过程中操作一些实物，例如，白板上的磁铁、回形针、牙签、记事贴、棋子或乐高积木。这种做法融入了三维视觉、触觉以及本体感觉（proprioception，使你感觉到身体的每个部分都在什么地方）等因素，可以令你更加深入地投入到正在研究的问题中。在研究队列、分组、协议、评级以及优先度等问题时，可以采用这种办法。但是别玩得太入迷了，这些物品的主要功能应该是帮助你进行调试工作。

要点

- 检查代码里面有没有常见的错误。
- 手工执行代码，以验证其是否正确。
- 通过绘图来解析复杂的数据结构。
- 在大型的纸张和白板上面，通过各种颜色的图示来演算复杂的问题。
- 在绘图的过程中操作实物，以便更深地投入到正在研究的问题中。

第 39 条：审读代码并与同事讨论

橡皮鸭技术（rubber duck technique）可能是本书里面的最有效办法了，你运用这个办法的次数越多，效果就越好。它要求你把代码的工作原理解释给别人听。解释到一半的时候，你通常就会惊叫："哎呀，我怎么这么傻，我知道问题出在哪了！"能喊出这句话，说明你并不笨，因为你找到的这个问题并不是那种单纯由于疏忽而造成的错误。在向同事解释代码的过程中，你大脑中活跃的那个部位与编写代码时所用的部位不同，这令你能够把当时发现不了的问题给找出来。在大多数情况下，你的同事其实只扮演了一个很小的角色，这就是该技术叫橡皮鸭的原因，也就是说，即便你对着橡皮鸭去解释，也依然能够有效地感觉到代码中的问题。（维基百科的 rubber duck debugging 词条里面有一张照片，真的就是一只小黄鸭坐在键盘上面。）

在你与同事之间还有一种更有意义的交流方式，那就是评审代码。这是一种更为正规的流程，它要求同事必须仔细阅读代码，指出其中的所有问题，包括代码风格、代码注释、API 的使用、设计以及逻辑等方面的错误。某些公司特别看重这项技术，要求开发者在把代码集成到产品分支之前，必须先做代码评审。Gerrit 等工具可以用来分享同事对代码所做的评论，GitHub 也提供了代码评论功能，这些都是很有用的，它们使你能够对评审意见做出回应，并把每一条意见的处理方式记录下来。

在评审代码的过程中，要注意礼节。不要把评审意见当成对你个人的批评，即便他们说得很难听，你也不要那样想，你应该把这当成提升代码质量的契机。要尽力应对每一条意见，即便有些意见说得不对，你也应该重视，因为这表示你没有把代码写清楚，从而引起别人误解。如果你邀请别人来评审你的代码，那你也应该去评审他们的代码，并且要用积极、专业而礼貌的方式去评审。不要把代码放在那里迟迟不评审、不要专挑一些小问题而回避更大的问题，也不要在评审过程中发生不愉快的事情，这些都会减损代码评审所带来的好处。

如果在某个涉及多方的算法里面出现了难以解决的问题，那么可以用角色扮演的方式来演练。比如，如果要调试某个通信协议，那你就来扮演其中的一方，让你的同事扮演另外一方，两个人轮流做出举动，试着破坏这个协议（或试着使其能够正常运作）。这个办法也可以用在其他一些领域，如安全问题（你们可以玩 Bob and Alice 的游戏[⊖]）、人机交互以及工作流等。你可以用某些物体来表示“编辑”操作，并在同伴之间传递这个标记。演练的时候，只要把意思表达出来就行，不用专门打扮，那样做显得有些过头。

要点

- 把代码解释给小黄鸭听。
- 做好代码评审工作。
- 通过角色扮演来对涉及多方的问题进行调试。

第 40 条：给软件添加调试机制

如果你能够告诉程序它正在接受调试，那么程序就有可能反过来主动协助你的调试

⊖ 密码学范例经常通过这两个角色之间的交流来进行讲解和演练。——译者注

工作。为此，你需要构建一套机制，以开启调试模式，并且要编写相应的代码来实现该模式。大家可以考虑采用下面这几种办法来使软件进入调试模式：

- 通过编译软件时所用的选项来决定软件是否进入调试模式，例如，编写 C/C++ 代码时，就可以定义 DEBUG 常量，并根据该常量的值来进行决策。
- 通过命令行选项来决定软件是否以调试模式运行。例如，Unix 系统的 SSH 守护进程命令（sshd）就提供了 -d 选项，其他很多程序也提供有类似的选项。
- 给进程发送信号（signal），令其进入调试模式。旧版的 BIND 域名服务器就是这么做的。
- 通过命令打开调试模式（这种命令可能不会写在开发文档中）。例如，通过某种很少见的按键组合来开启。（在某些版本的 Android 系统上面，连续单击七次 build number（版本号）菜单，就可以开启 USB 调试模式。）

为了防止自己不小心把调试版的软件发布到生产环境中，或是在生产环境里面无意间开启调试模式，你应该在调试模式启用的时候，给软件添加显著的标识。程序进入调试模式之后，我们可以通过编写程序代码来命令它做很多的事情。

首先，可以命令软件把执行过的动作记录下来，使得我们能够在发生某种事件时得到通知，并于稍后回顾各种事件的发生顺序（参见第 56 条和第 41 条）。

对于交互式程序以及带有图形界面的程序来说，我们还可以在屏幕上显示更多的信息，或是对已经显示出来的信息进行补充。比如，Minecraft 游戏在进入调试模式（参见图 5.2）之后，就会在屏幕上面叠加显示一些性能指标（每秒帧数、已经使用的内存量、CPU 负载）、玩家数据（坐标、方向、亮度）以及操作环境的配置情况（JVM、显示技术、CPU 类型）。此外，Minecraft 还能够在调试模式下查看整个游戏世界，它会以平面布局的方式，把游戏中的各种物体全都显示出来。在渲染程序中，我们可以把物体的每个面或贝塞尔曲线的每个控制点显示出来。在 Web 程序里面，我们也可以多显示一些有用的数据，比如，当鼠标悬停于对应的元素上面时，可以把该产品在数据库中的 ID 显示出来。

软件进入调试模式之后，可以启用其他一些命令。这些命令或许可以通过命令行界面、额外的菜单或某个 URL 来访问。我们也可以实现一些命令，用来展示并修改复杂的数据结构（如果改用调试器来显示并修改它们，那么操作起来会比较困难）、把数据转存到文件中以供后续处理，把程序切换到一种有利于排查错误的状态，或是执行我们在这部分所讲述的其他操作。

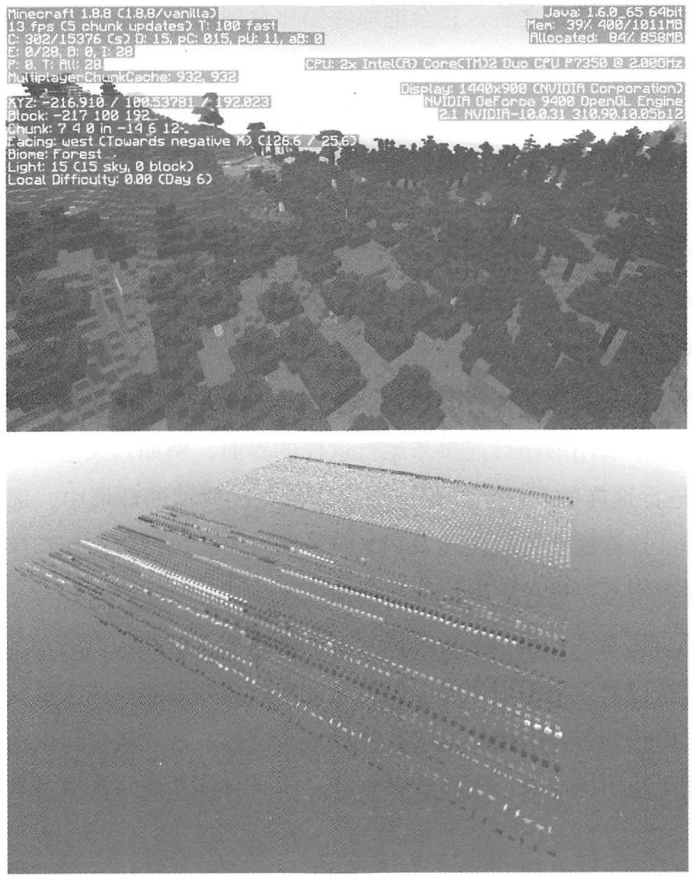


图 5.2 处在调试模式的 Minecraft 游戏所显示出的信息（上），以及调试模式之下的游戏世界（下）

调试模式还可以提供一个相当强大的功能，那就是令软件进入特定的状态。比如，在一个向导式的程序界面中，我们要对第 7 步进行调试。如果有了状态切换功能，那么调试工作就会变得简单一些，因为我们不用再执行前面 6 步了，而是可以直接通过捷径跳到第 7 步，程序可能会用一些较为合理的默认值来处理前面那 6 个步骤。对于游戏来说，也可以实现类似的调试功能，使得开发者能够直接升到比较高的级别上面（这样一来，你就不能在调试过程中以升级为借口来玩游戏了），或是直接拥有某项难以获得的能力。

进入调试模式之后，可以提升程序的透明度或简化程序在运行时的行为，使我们能够更加容易地锁定那些有待调试的故障。比如，有些程序本来应该是在后台静默执行的（这种程序在 Unix 系统中叫做 daemon（守护进程），在 Windows 系统中叫做 service（服

务)），然而开启了调试模式之后，它会把那些操作拿到前台来做，并且会在屏幕上输出一些信息。(Unix 的 sshd 守护进程，就是个典型的例子。) 如果程序平常会开启多个线程，那么现在就可以只用一个线程，以利于我们调试那些和并发无关的问题。此外，也可以把复杂而成熟的算法改为简单而原始的算法、把用来提升程序性能的那些辅助操作去掉、用同步的 API 来取代异步的 API，或是用内嵌的轻量级应用程序服务器或数据库来取代外部的产品。

运行在某些嵌入式设备或服务器上面的软件是没有用户界面的，对于这些软件来说，我们可以考虑在调试模式中展示更多的界面和接口，例如，可以提供命令行界面，以便输入调试所用的命令，并查看其运行结果。对于嵌入式设备来说，我们可以通过一个只在调试模式之下启用的串行连接来运行这样的界面，例如，某些数字电视就可以通过 USB 接口来进行这样的调试。如果应用程序运行在联网的环境中，那么可以开启一个像 libmicrohttpd 这样的小型嵌入式 HTTP 服务器程序，以便将应用程序的关键细节显示出来，并使得我们可以通过它来执行一些调试命令。

调试模式还可以帮我们模拟与外部因素有关的错误。这些错误通常都是出现概率很低的事件，需要通过繁杂的手段才能够模拟出来，然而如果能在调试模式里面提供一些命令，那么就可以较为容易地模拟出这些情况了，例如，可以通过调试命令来模拟网络随机丢包、数据无法写入磁盘、无线信号衰减、实时时钟功能错乱以及智能卡读卡器配置不当等情况。

最后，我们还可以在调试模式中执行罕见的代码路径。也就是说，我们不按照较优的那条路径来执行，而是修改程序的配置，令其走入另外一条执行路径。比如，如果有一个用来处理用户输入的内存缓冲区，其初始空间是 1KB，每次填满后扩充为原来的两倍，那么在调试这个程序时，我们就可以把缓冲区的初始空间从 1KB 降为 1 个字节，迫使程序更为频繁地进行重新分配，以观察并修复其在这个过程中所表现出的 bug。在其他情况下，也可以采用类似的手法来调试，如故意减小哈希表的容量，以观察程序在哈希表溢出时的行为，或是故意把缓存缓冲区设置得很小，以观察程序在缓存已满时，会采用什么样的策略来挑选并替换需要丢弃的缓存条目。

要点

- 给程序添加一个选项，令其能够进入调试模式。
- 添加相应的调试命令，使调试者能够操控程序的状态、记录其所执行的操作、

降低其在运行时的复杂程序、迅速在其用户界面之间跳转，并展示复杂的数据结构。

- 添加命令行、Web 以及串行连接等界面及接口，以便对嵌入式设备及服务器进行调试。
- 在调试模式下通过一些命令来模拟那些与外部因素有关的错误。

第 41 条：添加日志语句

日志（log）语句令我们能够追踪并理解程序的执行过程（参见第 56 条），这些语句通常会把消息发送给输出设备（例如，程序的标准错误端、控制台或打印机），或将其保存在某个地方（例如，文件或数据库中），以供稍后进行浏览及分析。有了这些日志，我们就可以通过查看其内容来寻找当前这个问题的根源。

有人认为：只有那些不会用调试器的人，才会通过日志语句来调试程序。这种说法在某些情况下是成立的，但是日志语句依然有几个地方要比调试器强，因此，这两种办法之间，应该是互补的关系。首先，我们很容易就能把日志语句放到程序中的关键点上，并且能够根据需要，把自己所关注的那一部分数据显示出来。相反，如果采用一般的调试器进行调试，那就必须按照程序的控制流来执行，并且要手工拆解那些复杂的数据结构。

此外，我们在调试器上面投入精力所获得的好处，可能很快就会消失，因为即便你把打印复杂数据结构时所用的那套配置保存到脚本文件中，其他的代码维护者也依然有可能无法看到或是很难获取这份文件。笔者碰到过一个项目，他们在分发源代码的时候，还要把调试器的脚本也一起包含进去。与之相反，日志语句所造成的效果是持久的，因此，我们更应该把在调试器上面所投入的精力，改投到日志语句中，以美化其输出格式，从而令我们更好地理解程序所执行的操作，进而提升调试工作的效率。

最后还有一个好处，那就是：只要日志语句编写得恰当（也就是说，要通过日志框架来写，而不是随兴地采用 println 来写），那么它所输出的内容，本身就可以进行过滤和查询。

大多数编程语言和开发框架，都有很多种日志程序库可供选用，你应该从中选择一款符合需求的库来使用，而不要自己去重新造一个库。需要记入日志的信息有：关键例程的入口和出口、重要数据结构的内容、状态的变化以及对用户操作所给出的回应等。请注意，这些日志语句不应该在生产环境中启用，以防程序性能由于日志数量过多而受

损。大多数日志库都提供了相应的接口，令我们可以根据信息的重要程度，从源头（也就是正在调试的程序）或目标（也就是用来记录日志消息的机制）处进行过滤。如果能够从源头控制住日志消息，那么它给程序性能带来的影响显然就会小很多，在某些情况下甚至可以完全消除。我们可以在应用程序里面实现调试模式，这样就能够根据需要来提升日志的详细程度了（参见第 40 条），此外，还可以利用与日志级别和范围有关的配置机制来详细调整自己想要看到的那部分日志条目。由于很多日志框架都提供了这样的配置机制，因此我们只需在应用程序里面直接使用它就可以了，而不用再重新开发一套。

可供选用的日志记录机制包括 Unix 的 syslog 库（参见程序清单 5.1）、Apple 的 ASL 系统日志机制（它的功能较为先进，参见程序清单 5.2）、Windows 的 ReportEvent API（参见程序清单 5.3）、Java 的 java.util.logging 包（参见程序清单 5.4），以及 Python 的 logging 模块（参见程序清单 5.5）。其中某些机制的接口，用起来不是特别容易，因此大家可以参考下面这些程序清单，并迅速地将其套用到你所写的代码中。如果你想寻找功能更为强大的工具，或是你所在的平台缺乏标准的日志记录工具，那么还可以考虑采用第三方的日志记录机制，例如，Apache 的 Log4j（适用于 Java）以及 Boost.Log v2（适用于 C++）等。

程序清单 5.1　用 Unix 的 syslog 接口来进行日志记录

```
#include <syslog.h>

int
main()
{
   openlog("myapp", 0, LOG_USER);
   syslog(LOG_DEBUG, "Called main() in %s", __FILE__);
   closelog();
}
```

程序清单 5.2　用 Apple 的 ASL 进行日志记录

```
#include <asl.h>

int
main()
{
   asl_object_t client_handle = asl_open("com.example.myapp",
         NULL, ASL_OPT_STDERR);
   asl_log(client_handle, NULL, ASL_LEVEL_DEBUG,
         "Called main() in %s", __FILE__);
   asl_close(client_handle);
}
```

非卖品，仅供非商业用途或交流学习使用

程序清单 5.3 用 Windows 的 ReportEvent 函数进行日志记录

```
#include <windows.h>

int
main()
{
   LPTSTR lpszStrings[] = {
      "Called main() in file ",
      __FILE__
   };
   HANDLE hEventSource = RegisterEventSource(NULL, "myservice");

   if (hEventSource == NULL)
      return (1);

   ReportEvent(hEventSource, // handle of event source
      EVENTLOG_INFORMATION_TYPE, // event type
      0,                  // event category
      0,                  // event ID
      NULL,               // current user's SID
      2,                  // strings in lpszStrings
      0,                  // no bytes of raw data
      lpszStrings,        // array of error strings
      NULL);              // no raw data

   DeregisterEventSource(hEventSource);
   return (0);
}
```

程序清单 5.4 用 Java 的 java.util.logging 包进行日志记录

```
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class EventLog {
   public static void main(String[] args) {
      Logger logger = Logger.getGlobal();
      // Include detailed messages
      logger.setLevel(Level.FINEST);
      FileHandler fileHandler = null;
      try {
         fileHandler = new FileHandler("app.log");
      } catch (IOException e) {
         System.exit(1);
      }
      logger.addHandler(fileHandler); // Send output to file
      logger.fine("Called main");
   }
}
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

程序清单 5.5 用 Python 的 logging 模块进行日志记录

```
import logging;

logger = logging.getLogger('myapp')

# Send log messages to myapp.log
fh = logging.FileHandler('myapp.log')
logger.addHandler(fh)

logger.setLevel(logging.DEBUG)
logger.debug('In main module')
```

此外，其他很多编程框架，也都有各自的日志记录机制。比如，如果你采用 lumberjack 协议来记录日志，那么在 node.js 环境中，就可以考虑选用 Bunyan 及 Winston 包；在 Unix shell 命令行界面里，可以通过 logger 命令把消息记入日志；在 Unix 内核（及设备驱动）里面，按照惯例，应该调用 printk 函数来记录消息。

如果程序运行在联网的嵌入式设备上面，而那些设备的可写文件系统又缺乏足够的空间来存放日志（例如，高端的电视机或低端的宽带路由器），那么可以考虑采用远程日志技术。我们可以运用该技术来配置嵌入式设备的日志系统，令其将日志条目发送给服务器，以便把这些条目存储在服务器里面。在配置 Unix 的 syslogd 时，可以通过下面这个办法，把与 local1 有关的所有日志条目，都发送给 logmaster 主机：

```
local1.* @@logmaster.example.com:514
```

如果你使用的编程环境没有提供日志记录机制，那就必须自己来做了。最简单的一种日志记录形式，是用 printf 这样的语句来打印日志消息：

```
printf("Entering function foo\n");
```

你或许觉得只要用 printf 式的语句把日志消息打印出来就行了，于是稍后会将其删掉或放在注释中。笔者建议你不要急着这么做，因为如果你删掉了这些语句，那么你编写它们时所花的功夫就白费了；如果你把这些语句注释掉，那么它们就不会在后续的代码修改过程中得到维护，继而会逐渐变得毫无用处。比较合适的做法应该是将其纳入条件语句中：

```
if (loggingEnabled)
   printf("Entering function foo\n");
```

除了采用打印消息的语句来进行记录，我们还可以通过其他一些手段来记录应用程序所执行的动作：

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 在GUI应用程序中，可以用弹出消息框。
- 在JavaScript代码里面，可以把消息写入控制台，并在浏览器的控制台窗口里观察结果。
- 在Web应用程序中，可以把想要输出的内容，以HTML注释或可见文本的形式，填充到显示程序运行结果的那个HTML页面里。
- 在无法修改程序源代码的情况下，可以试着使程序打开一份文件，该文件的名称就是你想要记录的那条消息，然后，用strace来追踪这个应用程序所执行的系统调用，并在其中寻找刚才那份文件的名称。

要点

- 通过日志记录语句来搭建一套可以持久维护的基础调试平台。
- 应该用现有的日志框架来记录，而不要去重新做一套框架。
- 根据你所关注的话题以及你想要记录的细节来对日志框架进行配置。

第42条：对软件进行单元测试

如果你在调试过程中，发现软件里面的某个问题无法通过单元测试表现出来，那就说明单元测试编写得不到位，或是项目中根本没有包含针对该问题的单元测试。为了把这样的问题隔离或捕获起来，我们可以考虑添加适当的单元测试，令问题得以暴露。

首先从最基本的平台做起。如果软件还没有开始使用单元测试框架，或是采用了一种不能直接支持单元测试的编程语言，那就下载符合需求的单元测试包（维基百科上面列出了很多单元测试框架），并对软件进行配置，使其可以使用该测试包来执行测试。如果项目中不存在早前写好的测试，那么此时应该调整构建配置（build configuration），把测试框架所对应的程序库包含进来，并且在应用程序的启动代码中添加几行语句，使其能够执行单元测试。接下来，对开发环境进行配置，令其在编译和提交软件的时候，能够自动运行单元测试。把单元测试平台添加进来，可以给项目带来很多好处，例如，可以使开发文档得以改进、使集体所有权意识得以明确、使重构工作变得顺利，并使集成工作变得简单。

然后，寻找那些可能与软件故障有关的例程，并针对这些例程编写单元测试，以验证其功能是否正常。我们可以用自顶向下或自底向上的思路来寻找这些待测试的例程

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

（参见第 4 条）。编写测试用例的时候，请试着不要去查看例程的实现方式，而是去查阅其接口文档，如果没有这种文档（很多情况下确实没有），那就看看其他代码是如何调用这些例程的。这样做可以使你不太会把编写产品代码时所用的错误思路，照搬到单元测试里面。写好的测试用例，应该提交到软件的版本控制系统里面，以便永久成为软件项目中的一部分。

现在举一个例子。我们来看程序清单 5.6 中的这个类，它可以在处理文本的过程中记录列的位置，并且能够把 tab 字符（制表符）的标准行为也考虑进去。这是个比较有难度的工作：在 20 世纪 80 年代，某些终端机无法正确处理含有 tab 字符的文本，于是出现了一些与屏幕输出有关的程序库，它们会采用相关的技巧来解决这类问题。程序清单 5.7 中的代码，用来启动程序清单 5.8 里面的 CppUnit 单元测试，以验证 ColumnTracker 类的功能是否正确。

程序清单 5.6　能够在处理文本的过程中记录列位置的 C++ 类

```
class ColumnTracker {
private:
   int column;
   static const int tab_length = 8;
public:
   ColumnTracker() : column(0) {}

   int position() const { return column; }

   void process(int c) {
      switch (c) {
      case '\n':
         column = 0;
         break;
      case '\t':
         column = (column / tab_length + 1) * tab_length;
         break;
      default:
         column++;
         break;
      }
   }
};
```

程序清单 5.7　启动 CppUnit 测试套件的文本界面所用的代码

```
#include <cppunit/ui/text/TestRunner.h>
#include "ColumnTrackerTest.h"

int
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
main(int argc, char *argv[])
{
      CppUnit::TextUi::TestRunner runner;

      runner.addTest(ColumnTrackerTest::suite());
      runner.run();
      return 0;
}
```

程序清单 5.8 单元测试的代码

```
#include <cppunit/extensions/HelperMacros.h>
#include "ColumnTracker.h"

class ColumnTrackerTest : public CppUnit::TestFixture {
   CPPUNIT_TEST_SUITE(ColumnTrackerTest);
   CPPUNIT_TEST(testCtor);
   CPPUNIT_TEST(testTab);
   CPPUNIT_TEST(testAfterNewline);
   CPPUNIT_TEST_SUITE_END();
public:
   void testCtor() {
      ColumnTracker ct;
      CPPUNIT_ASSERT(ct.position() == 0);
   }

   void testTab() {
      ColumnTracker ct;
      // Test plain characters
      ct.process('x');
      CPPUNIT_ASSERT(ct.position() == 1);
      ct.process('x');
      CPPUNIT_ASSERT(ct.position() == 2);
      // Test tab
      ct.process('\t');
      CPPUNIT_ASSERT(ct.position() == 8);
      // Test character after tab
      ct.process('x');
      CPPUNIT_ASSERT(ct.position() == 9);
      // Edge case
      while (ct.position() != 15)
         ct.process('x');
      ct.process('\t');
      CPPUNIT_ASSERT(ct.position() == 16);
      // Edge case
      ct.process('\t');
      CPPUNIT_ASSERT(ct.position() == 24);
   }

   void testAfterNewline() {
      ColumnTracker ct;
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        ct.process('x');
        ct.process('\n');
        CPPUNIT_ASSERT(ct.position() == 0);
    }
};
```

运行单元测试之后，有问题的例程就应该可以暴露出来了。如果测试并没有失败，那就需要扩大其覆盖范围，在个别情况下，还需要验证这些测试本身编写得是否正确。如果发生失败的测试用例有好几个，那就先关注位于依赖树底部的那些错误例程，也就是那些很少调用其他例程（或者说其他客户端）的例程。把有问题的例程修复好之后，应该再次运行测试，以确保所有的用例都能够通过。

为原来已有的代码编写单元测试，不是一件十分容易的事情，因为这与编写新代码时的情形有所不同：我们在编写新代码的过程中，通常会把测试用例一起写出来，并且会刻意用一种便于测试的方式来撰写这些新的代码；而面对已有的代码时，为了能够通过单元测试来锁定其中的可疑例程，我们可能需要对其进行重构（比如，把庞大的元素拆解成多个较小的部分，并尽量缩减例程之间的依赖关系），使得测试用例能够更为容易地调用这些例程（参见第 48 条）。与具体做法有关的技巧，不在本书的讨论范围之内，Michael Feathers 所写的《Working Effectively with Legacy Code》[⊖]（Prentice Hall，2004）一书，很好地讲解了这个话题。

要点

- 通过单元测试来检查可疑的例程，以便发现其中的错误。
- 为了提升测试效率，我们要选用合适的单元测试框架、要重构产品代码，使其便于接受测试，并且要使测试的执行得以自动化。

第 43 条：用断言进行调试

单元测试是一种寻找错误例程的重要工具（参见第 42 条），但仅凭它们自身，并不能完成所有的测试工作。首先，单元测试只会告诉你某个例程无法通过测试，它们并不会指出发生错误的具体位置。如果例程比较短，那我们只需要手工寻找错误就可

⊖ 该书中文版《修改代码的艺术》已由机械工业出版社引进出版，书号是 978-7-111-46625-3。——编辑注

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

以了，但如果要调试的是个复杂的算法，则很难把它拆解为多个自足的小例程。其次，有些错误并不是单独由某一部分代码所导致的，而是在多个代码块相互集成的过程中涌现出来的。高层的测试用例可以发现这些错误，但很少能够精准地锁定其原因。

这正是应该使用断言的时候。断言语句包含一条 Boolean 表达式，如果受测代码正确无误，那么该表达式的值必定为真（true）。如果它的值是假（false），那么断言就会失败，受测程序通常会以运行时错误的形式终止执行，并显示出与本次失败有关的数据。得知断言语句失败之后，通常可以用调试器来找出具体的错误位置。在代码中的关键地点放置断言语句，可以缩减你所要排查的范围。这种功效体现在两个方面。首先，你显然会关注发生断言失败的那一部分代码，此外，如果你所添加的某些断言没有失败，那你就可以把那些代码从可疑代码的范围中除去。

大多数语言都支持断言，有些是通过内置的语句来支持的，如 Java 和 Python，另外一些则通过程序库来支持，如 C 语言。为了尽量缩减由执行断言检查所带来的性能损失，许多编程环境都允许开发者开启或关闭断言功能。有些语言需要在编译期进行设置（例如，在 C 和 C++ 中，需要通过定义 NDEBUG 宏来进行切换），有些语言则需要在运行期进行设置（例如，在用 Java 虚拟机来运行程序的时候，可以通过 -enableassertions 或 -disableassertions 选项来开启或关闭断言）。在开发环境中运行代码时，我们一般会开启断言检查功能，而在生产环境中运行代码时，则要考虑开启该功能所带来的好处和损失，这需要根据具体情况进行权衡。

在对算法中的代码进行调试时，通常可以从前置条件（precondition，必须满足这些条件，算法才能够见效）、不变条件（invariant，算法在处理过程中，会令已经处理好的那部分数据满足这些条件）以及后置条件（postcondition，当算法根据规范把数据处理好之后，这些条件必然成立）这三个方面进行思考（参见第 3 条）。一般来说，在整个数据集里面，只有已经由算法所处理好的那一部分数据，才会满足不变条件，等到算法把全部操作都执行完毕，不变条件所涵盖的数据范围，就会变得和后置条件一样大了。

程序清单 5.9 中的代码可以体现出这种编程风格。为了在整型数组中寻找最大的元素，这段代码首先会把 max 变量设为整数型变量所能取到的最小值，然后依次与数组中的每个元素相比较，如果发现元素值比当前的 max 大，那就把该值赋给 max 变量，最后，max 变量中所保存的值，就是整个数组中的最大值。我们所要测试的前置条件是

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

该数组不为空，而且要保证算法一开始所选定的那个最小值，确实要小于或等于数组中的每一个元素。如果有人在修改了数组的类型之后，没有对 max 变量的初始值做出相应的调整，那么这个前置条件就有可能失败。算法的主体部分是一个大循环，该循环所维护的不变条件是：max 变量的当前值，大于或等于算法已经处理过的所有元素。如果这个大循环已经执行完了，那就意味着数组中的每一个元素都得到了处理，此时，算法的不变条件所涵盖的范围，就变得与后置条件相同，而这个后置条件，正是该算法所需满足的规范。

程序清单 5.9 采用断言语句来检查前置条件、后置条件与不变条件

```
class Ranking {
   /** Return the maximum number in non-empty array v */
   public static int findMax(int[] v) {
      int max = Integer.MIN_VALUE;

      // Precondition: v[] is not empty
      assert v.length > 0 : "v[] is empty";

      // Precondition: max <= v[i] for every i
      for (int n : v)
         assert max <= n : "Found value < MIN_VALUE";

      // Obtain the actual maximum value
      for (int i = 0; i < v.length; i++) {
         if (v[i] > max)
            max = v[i];
         // Invariant: max >= v[j] for every j <= i
         for (int j = 0; j <= i; j++)
            assert max >= v[j] : "Found value > max";
      }

      // Postcondition: max >= v[i] for every i
      for (int n : v)
         assert max >= n : "Found value > max";
      return max;
   }
}
```

除了像上面那样，用断言来检查（并表述）算法所执行的操作，还可以通过一些较为松散的用法来排解各种问题。例如：

- 在程序的开头放置断言，以验证 CPU 的架构属性（例如，其整数型数据的尺寸）是否符合要求。
- 在例程的入口点放置断言，以验证传入的参数是否具备正确的类型（如果你所使

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

用的编程语言不对参数类型进行检查），并确保它们具备有效（如不为 null）且合理的取值。

- 在例程的出口点，验证其结果是否正确。
- 对于经常受到调用或较为复杂的方法来说，可以在其开头与结尾放置断言，以验证类的状态是否能够保持一致。
- 如果要调用某个不应该出错的 API 例程，那么可以在调用完该例程之后，通过断言来确保这一点。
- 把软件所需的资源加载进来之后，用断言来验证该资源是否得到了正确的部署。
- 在对复杂的表达式进行求值之后，用断言来验证其结果是否具备应有的属性或合理的取值。
- 在 switch 语句的 default 分支里面，加入一条表达式取值为 false 的断言语句，以捕获未能由其他分支所处理的情况。
- 对数据结构进行初始化之后，用断言语句来验证该结构是否具备应有的取值。

总之，在调试的时候，可以通过断言语句来表达自己对代码的理解，并用它来测试可疑的代码。

如果你添加的断言语句，是为了表述代码所执行的操作，并用来防止以后有可能发生的问题，那么其中的大部分语句，都可以照原样保留在产品代码里面。然而要注意的是：如果你在调试过程中所添加的断言语句，能够发现一个有可能出现在生产环境中的问题，那么你就必须用更为健壮的错误处理代码来替换这些断言语句，这样才算真正完成调试工作。例如，你应该用错误处理代码来对用户所输入的内容或其他一些不为你所控制的资源进行验证，并确保某些可能会出错的 API 能够得到正确的执行。此外，我们也可以同时采用断言和单元测试代码来验证某个例程，这里之所以要提到单元测试，是因为这些测试可以自动得到执行，而且还有助于提升产品代码的测试覆盖度。

要点

- 用一些与单元测试相互补充的断言语句来更加精准地锁定代码中的错误。
- 通过断言语句来调试复杂的算法，以验证其前置条件、不变条件与后置条件是否成立。
- 在调试过程中，用断言语句来表达自己对代码的理解，并用其测试可疑的代码。

非卖品，仅供非商业用途或交流学习使用

第 44 条：改动受测程序，以验证自己的推想

我们总是把那种随意修改程序并观察其行为的做法，贬称为 hacking。然而如果你是在深思熟虑之后才进行实验的，那么这种做法就可以帮助你验证之前所做的假设，并且可以使你更深入地了解正在调试的系统及其底层平台。对于那些自身品质不是特别高的产品代码来说，这种修改尤其有用，因为它可以帮你填补文档及 API 中的空缺。

调试系统的时候，可能会遇到下面这样的问题。我们只需要修改代码，就能够轻松地回答这些问题：

- 例程的某个参数能不能为 null？
- 如果这个用来表示毫秒的变量，其值大于 999，那么代码还能不能正常运作了？
- 如果程序进入该例程的时候，发现有人正在持有某一把锁，那它会不会给出警告？
- 这些方法之间的调用顺序，与当前要解决的问题有没有关联？
- 有没有比现在用的这个 API 效果更好的其他 API？

改动代码之后，我们通常会通过观察程序行为、查看日志记录，或是在调试器中运行程序等手段来验证自己的推想是否成立。

有一种做实验的方式，是修改代码中的表达式和值，把那些在程序运行时才能够定下来的表达式替换成常量表达式。比如，你可以把某个正确的常量传给例程（或是令例程总是返回这个常量），然后看看你要调试的那个问题有没有消失。你也可以故意把错误的值传给例程（或是将其设为例程的返回值），然后看看程序的故障是不是由这个值所造成的。此外，还可以设置极端的参数值，使得某个较为微妙或罕见的问题（如性能下降），能够更加容易地表现出来（参见第 17 条）。

还有一种做实验的方式，是改用其他的办法来实现程序中的某个功能，然后看看修改后的代码是否正确。我们把可能有错误的那一部分代码，替换成另外一段比较好的代码，然后看看这样做能否解决问题。比如，Microsoft Windows 提供了 5 种以上 API，用来获取屏幕上某个字符串的宽度，然而它并没有告诉我们怎样在其中选择最为合适的那个 API。如果你遇到的问题是文本对不齐，那么可以把 GetTextExtentPoint32 换成 GetTextExtentExPoint，然后看看能否对齐。如果你怀疑程序中的错误与某些例程之间的调用顺序有关，那就试着按照另外一种顺序来调用这些例程。此外，还可以尝试对代码进行大幅度的简化（参见第 46 条）。

要点

- 手工设定代码中的某些值，以验证哪些取值是正确的，哪些取值是错误的。
- 如果找不到修改代码的正确方法，可以试着用其他的方式来实现它。

第 45 条：尽量缩小正确范例与错误代码之间的差距

有时我们可能会对比两份代码，一份是当前正在调试的代码，另一份是一段可以正常运行的范例代码。这种情况通常发生在调试复杂 API 或调试算法的时候。能够正常运作的那个范例，可能是从 API 文档、问答网站（参见第 2 条）、开源软件或教科书里面抄录的，将该范例与自己的代码相对比，可以帮助你找到有错的地方。笔者在这里所要讲解的办法，是通过修改源代码来进行对比，此外，也可以从运行时的行为入手来进行对比（参见第 5 条）。

如果你要参照范例代码来调试自己所写的代码，那么必须首先保证这段范例代码能够编译，并且可以正常地运行。若是做不到这一点，那就说明你所遇到的问题，可能不是由自己所写的代码造成的，而是由于你所在的工作环境（编译器、运行时环境以及操作系统）有问题，或者你误以为某个 API 或算法能够实现你想要的功能，但实际上却实现不了。此外，还有一种概率比较低的情况，那就是第三方的代码里面有 bug。

对你要参考的这份范例代码做了验证之后，现在可以有两种办法来调试自己的代码。这两种办法都是要逐渐缩减范例代码与错误代码之间的差距。从理论上来说，如果你能把两者之间的差距消除，那么你的代码就可以像范例代码那样正常地运作了。

第一种办法是根据自己所写的代码来逐步修改并编译范例代码。如果要调试的是那种简单而自足的代码，那么这个办法的效果就很好。我们从自己所写的代码里面拿出来一些内容，将其一点一点地放到范例代码中，每次修改完范例代码之后，就去看看它的功能是否正确。如果发现加入了某个内容之后，范例代码无法正常运作，那就说明程序中的错误，是由这个内容所引发的。

第二种办法是根据范例代码来删改自己的代码，使其与范例相符。如果你自己的代码中有着很多依赖关系，不太容易单独拿出来进行操作，那么这个办法就比较合适。你需要逐步删减或调整自己的代码，使其与范例代码相匹配。这个过程要一点一点地来做，每次修改完之后，看看自己的代码是不是依然有问题。如果在删掉或修改某个内容之后，代码突然可以正常运行了，那就说明你要解决的问题，与刚才删改的那个内容有关。

要点

- 逐渐缩减你自己的代码，使其与范例代码相符，或是逐渐修改范例代码，使其与你自己的代码相符，这两种办法都可以用来查找程序出错的原因。

第 46 条：简化可疑代码

复杂的代码是很难调试的，因为其中有很多条执行路径需要考虑，而且还有相互纠缠的数据流需要厘清，这些都会增加我们排查错误所需的工作量。为了使代码调试起来容易一些，我们通常需要对其进行简化。这种简化可能是临时的（参见第 17 条），也可能是永久的，前者是为了凸显程序中的错误，后者则是为了彻底修复该错误。在开始大幅度简化代码之前，首先要保证代码可以适当地回复到早前的版本。你应该把需要修改的每一个文件，都纳入版本控制系统（参见第 26 条），而且还要能够把代码恢复到初始状态，为了做到这一点，你可以考虑在自己的分支上面进行修改。

如果采用临时修改的办法，那么通常需要大幅度地删除代码。在保证程序依然会出错的前提下，你需要尽可能多地删减代码，以便将可疑代码的数量缩至最小，从而能够更加容易地找出其中的错误。一般来说，我们首先会删掉一大块代码，或是删掉对某个复杂函数的调用操作，然后编译程序，并查看其运行结果。如果程序这次依然有错，那就继续删除，如果程序没错，那就把刚删掉的这部分代码复原，下次试着从里面少删除一些代码。如果删去某一部分代码之后，程序就不出错了，那么你完全有理由认为：程序的错误与删去的代码之间有着某种联系。将可疑代码的范围缩减到一定程序之后，可以采用另外一种做法来排查问题，也就是设法删掉数量极少的内容，使得程序在删掉这部分内容之后，变得可以正常运行。

你可以把每一步的删除操作都记录在版本控制系统中，然而采用编辑器自身的功能来做，通常会更快一些。用编辑器打开代码，每删掉一段，就保存一次。如果删掉之后程序依然出错，那就继续删除；如果错误消失，那就通过撤销功能来还原上一步所删的代码，然后选择一个比上次小的范围，继续删除代码。你可以按照二分搜索算法，有规律地执行这个过程。

不要急着把你想排除的那些代码直接注释掉，因为如果些代码里面本身就含有注释，那么这种多重嵌套的注释可能会引发问题。对于支持预处理程序的语言来说，可以把这段代码放在带有条件的预处理块中。

```
#ifdef ndef
  // code you don't want to be executed
#endif
```

对于其他语言来说，可以把需要禁用的那些代码，放在 if(false) 条件语句块里面。

有时不一定非要把代码删掉，而是可以通过对其进行调整来实现简化。比如，只需要在 if 或 loop 语句块的条件开始处多写一个 false 值，就可以确保语句块里面的代码不会得到执行了。例如，下面这段代码：

```
while (a() && b())
  someComplexCode();

if (b() && !c() && d() && !e())
  someOtherComplexCode();
```

可以分步骤简化为：

```
while (false && a() && b())
  someComplexCode();

if (false && !c() && d() && !e())
  someOtherComplexCode();
```

在其他一些情况下，我们可以永久地简化复杂语句，以降低调试的难度。例如，下面这个语句：

```
d = s.client(q, r).booking(x).period(y, checkout(z)).duration();
```

像这样拼接成一长串的语句，不妨称为 train wreck 语句，因为它很像火车遭遇事故之后的一串车厢。由于我们无法明确地看出其中每个方法的返回值，因此调试起来是很困难的。我们可以添加委托方法（参见第 48 条）来解决这个问题，也可以像下面这样，将表达式拆分成几个小的部分，并把每一部分的结果放到一个临时变量中。

```
Client c = s.client(q, r);
Booking b = c.booking(x);
CheckoutTime ct = checkout(z);
Period p = b.period(y, ct)
TimeDuration d = p.duration();
```

这样修改之后，就更容易在调试器里面看出每次方法调用的结果了，此外，还可以更为方便地添加相应的日志语句。由于类型或变量采用了有意义的名称，因此代码也变得更加易读。性能问题是无需担心的，因为当前的编译器都明白应该怎样消除不必要的临时变量。

另一种值得考虑的简化方式，是把大函数拆成多个小的部分。这样做给调试工作所带来的好处主要在于：我们可以通过单独测试每一个部分来找寻函数中的问题（参见第 42 条）。还有一个好处是：在这个过程中，我们可以更好地理解受测代码，并厘清各部分之间的关系，从而将其中没有必要的那些交互操作删去。这两项优点，有助于我们找到解决问题的办法。

还有一种幅度更大的永久简化方式，那就是弃用某些复杂的算法、数据结构或程序逻辑。之所以能够这样做，是因为我们所要寻找的错误，可能正好位于这些复杂的代码中，而这些代码，其实完全可以改用其他方式来代替。下面举几个典型的例子：

- 从前的某些优化算法，在当前这个处理器速度很快的时代，已经没有必要使用了。在 3MHz 的 VAX 计算机上，用户或许能够感受到响应时间为 500 毫秒和 5 毫秒的两个算法所体现出的区别。然而对于频率为 3.2GHz 的 Intel Core i5 CPU 来说，这两种算法的响应时间分别变成了 500 微秒与 5 微秒，用户已经不可能感受到其中的差别了。在这样的情况下，如果要处理小型的固定数据集，那么完全可以改用更为简单的算法来做。
- 当今的内存容量、磁盘空间以及网络带宽都比较丰富，因此我们很少需要像原来那样，采用复杂的位运算来把数据塞进某种二进制结构中，而是可以直接采用编程语言自带的 integer（整型）或 Boolean（布尔）等变量类型来表示数据。其他一些复杂的数据压缩方案，也可以做类似的处理。
- 硬件技术发生改变之后，我们就没有必要再使用以前的某些优化算法了。比如，操作系统的内核曾经包含一种复杂的电梯算法，用来优化磁盘磁头的移动方式。然而对于当前的磁盘来说，因为我们没有办法知道某个数据块在盘片中的具体位置，所以这种算法是派不上用场的，此外，由于固态硬盘的寻道时间几乎为 0，因此，就更不需要使用这些复杂的算法和数据结构了。
- 某些容易出 bug 的算法，可以改用编程框架中的库或成熟的第三方组件来实现。比如，如果你想编写一个时间复杂度为 $O(n)$ 的算法，以寻找容器里面各个元素的中位数（median value），那么你所写出来的算法，可能会包含很多微妙的 bug。在这种情况下，你可以改用 C++ 的 std::nth_element 函数来查询中位值，以避开这些问题。从更大的层面上来说，如果你所使用的是某一款专用的数据存储及查询引擎，而该引擎又有着一些 bug，那么可以考虑用关系型数据库来取代这个引擎。
- 某些旨在改进程序性能的复杂算法，其实是小题大做。只有当性能分析或其他

一些分析的结果确实要求我们对某些关键代码做出优化时，我们才需要着手考虑这些算法。程序员有时会忽视这条原则，从而无谓地创造出一些过于精巧且过于复杂的算法。如果你能弃用这些代码，那么自然就不需要再担心其中的bug了。

- 与过去相比，当前的用户更喜欢简洁的交互方式。我们从前需要用复杂的代码来展示烦琐的对话框，并在其中放置很多种可以调节的参数，而现在，只需要用较为简单的代码来展示几个精心挑选的参数即可，至于其他的参数，则不妨设为合理的默认值。

要点

- 有选择地删除大段代码，使错误变得更加突出。
- 把复杂的语句或函数拆成多个小的部分，以便单独监控或测试其功能。
- 考虑弃用那些可能会出bug的复杂算法，并改用简单一些的算法来实现。

第47条：将可疑代码改用另外一种编程语言来写

如果你发现某个问题无论如何都解决不了，那么可能要采取一些幅度更大的措施。其中一个措施，是采用另外一种语言来改写这些有问题的代码。在更为优秀的编程环境中，我们可以采用更好的工具来理解问题，并创建出可行的初步解决方案，以便完全绕过该bug，或找到并修复它。（这条原则在并发问题上面的运用请参见第66条。）

新选的这门编程语言，写起代码来应该比当前用的这种语言更加方便。比如，如果想更加容易地评估复杂的交易策略，那么就应该改用R、F#、Haskell、Scala或ML等支持函数式编程的语言。此外，还可以通过编程语言的程序库来获得更多的便利。在某些情况下，这样做所带来的好处是很大的，大到我们几乎不可能退回到原来那些功能不太丰富的语言，比如，统计方面的计算工作，就是应该改用R语言来完成才好。再举一个例子：如果你要在动态分配的元素集合中，执行一些棘手的字符串操作，那么可以考虑用C++或Python语言来改写原有的C语言代码。采用表意能力更强的语言编写程序，可以令实现代码变得更为简洁，从而减少出错的概率。

还有一个与调试工作有关的语言特性也值得注意，那就是：你所选用的编程语言能不能较为方便地查看代码的行为，比如，是否能够通过渐进的方式来构建这些代码。从

这一方面看，脚本语言的优势是比较大的，因为这些语言可以按照读取—求值—输出的流程来反复循环（read-eval-print loop，REPL）。如果你是通过 Unix 工具加管道的方式来实现算法的，那么可以考虑先验证当前这个阶段的输出是否正确，然后再添加下一个阶段。此外，如果你所使用的开发系统没有提供合适的调试、日志记录或单元测试框架，那么可以考虑迁移到更为先进的实现环境中，以便采用更为强大的机制来解决你正在调试的这个问题。当你在小型的嵌入式设备上面，采用功能较少的开发工具进行调试时，尤其应该考虑改用另外一套开发环境。

如果新编写的代码能够正常运作，那我们可以考虑采用两种办法来解决早前遇到的问题。第一种办法是在软件项目里面采用新代码实现相关的功能，并把原来的代码删掉。如果新旧两种编程语言结合得比较好，那么改编起来是很容易的。例如，对于那种只有少数几个纯数据形式的参数，而且返回值也比较简单的 C++ 函数来说，我们很容易就能在 C 语言里面调用它。此外，也可以把新写好的这份实现代码，当成独立的进程或微服务来调用，当我们不太在乎调用所引发的开销时，这种做法是可行的。

第二种办法，是参照新代码来修正旧代码。我们可以观察这两份代码在行为上的区别（参见第 5 条），或逐渐缩小这两份代码之间的差距，以便使 bug 能够浮现出来（参见第 45 条）。前一种做法要求我们在两种实现方式所用的变量及例程的返回值之间进行比较，而后一种做法则要求我们通过逐渐试错来寻找正确的实现方式。

要点

- 采用另外一种表达能力更强的语言来改写那些难以修复的代码，以减少可能出现问题的语句数量。
- 把有 bug 的代码移植到更好的编程环境中，以便采用更为强大的调试工具来解决其中的问题。
- 把新的实现方式写好之后，我们可以采用这种方式来完成原有的功能，或是参照它来修改旧的代码。

第 48 条：改善可疑代码的可读性与结构

混乱而糟糕的代码，很容易滋生 bug。我们应该清理这些代码，以便发现并修复其中的 bug。然而在开始清理之前，首先必须保证自己确实有足够的时间和相应的权利来

做这件事。如果你为了修复某个 bug，想要一下子改动四千多行代码，那么是不会有人轻易赞同你这种做法的。你至少应该把装饰性的调整，与代码重构及真正的 bug 修复工作分开，并且要分别进行提交。在某些环境中，最明智的办法是和同事相互商量，看看应该怎样协调前面两种改动。

调整代码时，应该首先由空白入手。从最底层来说，我们要确保操作符及保留字两边的空格使用方式，能够在整个项目里面保持一致，而且要与当前团队所采用的风格相符，以帮助我们发现语句和表达式中某些微妙的错误。在稍高一些的层面上，我们要确保缩进的空格数在整个项目中都是相同的（一般来说，缩进 2、4 或 8 个空格）。如果代码缩进得比较整齐，那么其控制流就会显得更加清晰。跨越多行的语句尤其值得注意，对这种语句进行适当的缩进，可以令我们看出其中那些复杂的表达式及函数调用操作写得是否正确。从最高层来说，我们要添加适当的空白，以帮助阅读者理解代码的含义。比如，可以通过多添加空格的办法来把相似的一组表达式对齐，使得其中有问题的地方得以凸显，也可以通过添加空行的办法来把代码分成多个代码块，使得每个代码块都能够包含一些逻辑上较为接近的语句，以帮助阅读者更好地理解代码的结构。总之，代码的样子要和它的功能相对应，这样才能令我们更容易看出其中的问题。如果手动调整代码格式所需的工作量实在太大，那么可以考虑通过 IDE 或 clang-format 及 indent 这样的工具来自动进行调整。

对代码格式进行调整，可以改善其外观，然而它的功效仅止于此。把格式修整好之后，接下来还应该考虑是否需要重构（refactor）。所谓重构，是指在维持代码功能不变的前提下，改善其结构。为了修复代码中的问题，我们要么把代码重构成更为有序的结构（这种做法接近于重写，参见第 47 条），要么就对代码进行规整，使得其中的错误变得更加突出。下面列出一些常见的代码异味（code smell），这些异味有可能会掩盖代码中的错误，我们可以通过适当的重构来解决这些问题。大部分例子都源于 Martin Fowler 所著的《Refactoring:Improving the Design of Existing Code》（Addison-Wesley, 2000）一书。与重构有关的详细内容，请参阅这本经典的教程。

重复的代码（duplicated code）可能会引入 bug，因为如果你只对其中的几份代码做了改进与修复，而没有更新其他的副本，那么未更新的那些代码就有可能出现问题。我们应该把这些重复的代码全都放到一个常用的例程、类或模板里面，这样只需要在一个地方做出修改，就可以在整个程序中见效。如果 bug 确实是因为这些相似的代码没有同步更新而导致的，那么在把刚才说的那个公用例程提取好之后，就可以逐个删除原来

的副本了。我们在删除每一个副本时，都将该副本与提取好的公用版本相比对，以确定 bug 究竟是由哪一个副本所引起的。

重复的代码也有可能隐藏在 switch 语句里面，这些语句通常会根据某个代表数据类型的值来改变程序的流向。当我们在某些 switch 语句里面添加新的判断情况时，很容易就会忘记编写相应的 case 从句，从而导致程序悄无声息地出现问题。要想解决这个问题，有一种简单的办法是添加 default 从句，并在程序执行到该分支时打印 log 消息，以记录这一错误。更好的一种办法则是重新调整代码结构，以彻底消除 switch 语句。我们通常会把与每个 case 分支有关的行为，移入各子类的对应方法中，并且对这个多态方法进行调用，以替换原来的 switch 语句。此外，也可以用状态对象来替换原有的 switch 语句，并把各 case 分支中的行为分散到状态对象的各个子类中。

还有一个与之有关的代码异味，叫做霰弹式修改（shotgun surgery），也就是那种修改一处代码会影响多个方法与字段的情况。你所要调试的 bug，有可能恰恰就是因为有人忘了对受影响的方法或字段进行调整。我们应该把受影响的这些方法与字段全都放到同一个类里面，以确保它们总是能够彼此协调地进行变化。这个类可以帮助我们缩减所需改动的代码范围，它可以由已有的类、新建的类或某个内部类（也叫做嵌套类）来充当。

另外一种代码异味也容易引发代码不一致的问题，这就是数据泥团（data clump），它是指那些经常同时出现的数据对象。我们应该把这些数据放到某个类中，并改用那个类的对象来表示某些参数及返回值，这样不仅能够对原来众多的数据对象进行归并，而且还可以防止由于忘写某个数据对象所引发的 bug。

如果你采用 integer 或 string 等编程语言的基本类型值（primitive values）来表达像货币、日期或邮编这样的复杂概念，那么就有可能在操作这些值的时候不小心犯错。比如，如果你用整数来表示货币值，那就有可能把两种不同的货币值直接相加。我们应该用类来表示这样的概念，并且用类的对象来替换代码中的基本类型值。与之类似，基本类型的数组，也应该改成链表或大小可变的 vector 等容器，这样可以帮你修复那些与数组尺寸管理有关的错误。更进一步的做法，是专门针对时间、质量、力、能和加速度等具体的物理概念来定制特别的类，而不采用原始的浮点数类型来表示它们的值。这样做令我们能够以适当的方法来表示这些类之间的结合关系（例如，$F=m\times a$，力 = 质量 × 加速度），从而避免那种类型错乱的赋值行为，如把 apple(苹果）类型的值赋给 orange(橘子）类型的变量。

非卖品，仅供非商业用途或交流学习使用

纷繁的接口会令代码结构变得混乱，从而容易滋生 bug，比如，每个类都各自支持一套不同的方法，而这些方法的名称、参数顺序及参数类型也都各有区别。面对这种情况，我们应该通过重命名及调整顺序等做法来使这些方法的名称及参数顺序彼此相似，并且通过删除和移动方法等手段来使各个类所拥有的方法也彼此相似。等到这些类都具备类似的新接口之后，我们或许还能够发现其他一些重构的机会，例如，可以把相似的接口提取到超类里面。

过长的例程是很难阅读和调试的（参见第 46 条），我们应该将其拆分为多个小块，并对复杂的条件进行分解，通过对其他例程所进行的调用来表达这个复杂的逻辑。如果某个方法由于使用了过多的临时变量，从而导致拆分起来很困难，那么可以考虑将其改为方法对象，并把临时变量转换为该对象的字段。

如果代码片段之间依赖得过于紧密，那么就有可能把某些错误的交互行为掩盖起来，这些行为会破坏本来应该成立的不变条件以及程序的正常状态。我们应该通过移动方法及字段等办法来尽量打破这种依赖关系，如果类之间确实要有依赖关系，那就尽量确保这种关系是单向而非双向的。过长的方法调用链，可能会向客户代码暴露一些它们无需访问的内容，这也有可能引发程序错误。我们可以通过引入委托方法来拆解这样的方法调用链。比如，下面这个表达式就显得有些长：

```
account.getOwner().getName()
```

通过引入名为 getOwnerName 的委托方法，我们可以将其改为：

```
account.getOwnerName()
```

还有一个令人意想不到地方，也可以进行重构，那就是代码中的注释，因为有些人喜欢用注释来掩盖一些不易理解或质量不佳的代码。一般来说，我们应该把写在注释中的代码块提取到方法里面，并给这个方法起一个适当的名称，以反映这些语句所要表达的意思，然后在原来写有大段注释的地方调用这个方法，以求简化程序的代码，并凸显其中的逻辑错误。此外，也可以用断言（参见第 43 条）来更加有效地表达注释中所提到的前置条件，因为只要条件得不到满足，断言语句就会立刻失败，从而引起我们的注意。

最后，还要注意删掉那些永远执行不到的代码（dead code），以及臆想出来的通用代码（speculative generality）。删除无用的代码和参数、缩减无用的类体系、把仅有一个客户代码会调用的类进行内联，并给那些名称特别抽象的方法重新起名，以反映其真实功能。这些举措都可以使 bug 无处藏身。

非卖品，仅供非商业用途或交流学习使用

要点

- 用连贯的格式来修整代码，以凸显其中的错误。
- 对代码进行重构，以消除那些品质不佳或毫无用处的复杂结构，使潜藏在其中的 bug 得以暴露。

第 49 条：要清除 bug 的根源，而不仅仅消除其症状

令人奇怪的是，有很多开发者都急于采用应付的办法来掩盖 bug，并试图以此化解程序中的问题。下面举几个采用条件语句来“修复”bug 的例子：

❑ 避免引用空指针：

```
if (p != null)
  p.aMethod();
```

❑ 绕开除数为 0 的问题：

```
if (nVehicleWheels == 0)
  return weight;
else
  return weight / nVehicleWheels;
```

❑ 把变量的取值强行调整到合理的范围内：

```
a = surfaceArea()
if (a < 0)
  a = 0;
```

❑ 对遭到截断的姓氏进行修正：

```
if (surname.equals("Wolfeschlegelsteinha"))
  surname = "Wolfeschlegelsteinhausenbergerdorff";
```

在上面那些语句中，有几个例子确实可以找到合理的借口。然而你如果在没有探查根源的前提下，就仅仅通过这些条件语句来绕开某个导致程序崩溃的 bug、某种异常或某个错误的结果，那么这种做法就说不过去了。（笔者见过一个最为极端的例子，是在 C 语言里面设法把字符串中混入的控制字符给删掉，然而这些控制字符究竟是怎么混进来的，则没有人清楚。）

采用权宜的代码来绕开 bug 是一种很不好的做法，因为这会造成许多问题：

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 通过避开某些功能代码来“修复”现有的 bug，有可能会引入更为微妙的新 bug。
- 由于你并没有清除掉该 bug 的根源，因此程序里面可能还会存留一些不那么明显的症状，而且有些 bug 以后或许会以另外一种形式表现出来。
- 程序的代码会无谓地复杂起来，从而变得难以理解和修改。
- 由于这种权宜的“修复”方案把症状给掩盖起来了，因此 bug 的根源就会变得更难查找。比如，当程序发生崩溃时，你本来可以据此去寻找 bug 的根源（参见第 55 条），但你却选择了用临时代码来绕过这个 bug。

由此可见，只掩盖 bug 症状而不清除其根源的做法，是一种看似轻松但实际上很费事的举动，它会令你的程序背负技术债务（technical debt）。

还有一个与之相关，但不那么严重的做法也值得注意，那就是：在本来应该以简单的方式提出通用解决方案（参见第 46 条）的场合，却给出了一种只能应对特例的修复办法。比如，下面这段代码本来应该把角调整到 0 至 $2\times\pi$ 这个范围内，但是它却只能应对很小的一个范围，而无法正确处理小于 $-2\times\pi$ 或大于 $4\times\pi$ 的诸多情况：

```
if (angle < 0)
  angle += Math.PI;
else if (angle > 2 * Math.PI)
  angle -= Math.PI;
```

我们可以将其改成下面这样：

```
while (angle < 0)
  angle += Math.PI;
while (angle > 2 * Math.PI)
  angle -= Math.PI;
```

但是这样写太复杂了，而且当角的值特别大或特别小时，计算起来较为缓慢，并且不够精确。我们应该改用下面这种计算余数的做法来把角的值调整到 0 至 $2\times\pi$ 之间：

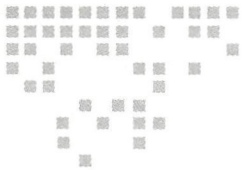
```
angle = angle - 2 * Math.PI * Math.floor(angle / (2 * Math.PI)));
```

要点

- 不要采用临时代码来绕开程序的表面症状，而是要查找 bug 的深层原因并加以修复。
- 尽可能采用通用的办法来处理复杂的情况，而不要只修复其中的某些特例。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



Chapter 6 第 6 章

编译时的调试技术

如果把源代码转变成可以在 CPU 或抽象虚拟机（如 JVM）上面运行的字节码，那我们就能够充分地观察程序的行为及其对运行结果所造成的影响。这两个方面都有助于我们详细探查当前正在调试的问题。

第 50 条：对生成的代码进行检视

代码通常要经过一系列的转换，才能够最终变成处理器指令，在这个转换的过程中，它会以不同的形式而存在。例如，C 或 C++ 文件可能需要先接受预处理，然后编译成汇编语言，接下来，再组合到一份目标文件中；Java 程序会编译成 JVM 指令；用来生成词法分析器和语法分析器的那些工具（如 lex、flex、jacc 和 bison），会把输入给它们的内容编译成 C 或 C++。我们可以通过各种命令及选项来观察这个转换过程，并查看该过程所生成的中间代码，以便形成一些对调试工作很有帮助的想法。

现在我们来考虑经过预处理的 C 或 C++ 源代码。运行 C/C++ 编译器的时候，只需要添加一个选项，就可以把预处理之后的源代码显示出来，在 Unix 系统中，这个选项是 -E，对于 Microsoft 的编译器来说，这个选项是 /E。（在 Unix 系统中，我们也可以通过 cpp 命令来直接运行针对 C 语言的预处理程序。）如果编译器显示出来的代码有很多行，那么可以把结果重定向到文件中，以便稍后能够在编辑器里面仔细检视其内容。下

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

面举一个例子，告诉大家应该怎样查看预处理之后代码并找寻其中的错误。

```
#define PI 3.14159265358979323846264338327950;
double toDegrees = 360 / 2 / PI;
double toRadians = 2 * PI / 360;
```

用 Visual Studio 2015 编译器来编译上述代码，会产生一个看上去（可能）比较奇怪的错误：

```
t.c(3) : error C2059: syntax error : '/'
```

如果你去查看编译器预处理之后的结果，就会发现，上述错误是 / 左侧的分号引起的，它之所以会出现在这里，是由于我们在定义宏的时候，多写了一个分号。

```
#line 1 "t.c"

double toDegrees = 360 / 2 / 3.14159265358979323846264338327950;;
double toRadians = 2 * 3.14159265358979323846264338327950; / 360;
```

第三方的头文件可能含有复杂的宏及定义，这些内容在展开时或许会出错，而刚才所讲的这个技巧，则特别适合用来调试这些错误。但是，如果展开之后的代码太多（通常情况下确实很多），那就很难在其中找到有问题的那部分代码了。这种情况下，有一个办法是在无法编译的那一行源代码附近，寻找某个与宏无关的标识符或字符串（对于刚才那个例子来说，我们可以搜寻 toRadians），甚至还可以在你所关注的那部分源代码附近，添加一条纯粹起标识作用的声明语句，以便能够在预处理之后的代码里面，迅速地跳转到这个地方。

我们也可以在经由预处理或其他方式所生成的代码中删除 #line 指令，以便更好地寻找其中的错误。预处理后的文件里面所包含的 #line 指令，能够使编译器明白：它当前所读到的这行代码，在其原始文件里面究竟位于哪一行上面。有了这个信息之后，编译器发现错误时，就会精确地指出错误代码在其原始文件中的位置，而不会给出这行代码在预处理之后的文件中所处的位置。然而，我们有的时候就是想要知道错误代码在预处理后的文件里所处的位置，而不想在错误消息里面看到该代码在原始文件中的位置。于是，可以通过 -P（适用于 Unix 系统）或 /EP 选项（适用于 Microsoft 的编译器）来命令编译器不要输出 #line 指令。

在其他情况下，可能还需要查看生成之后的机器指令。这也可以帮助我们发现一些平常找不出来的低级错误。比如，我们可能是在看了机器指令之后，才发现自己用错了某个操作符、写错了某个数值类型，或是漏掉了某个括号或某条 break 语句。查看机器

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

码，还可以帮我们调试一些与底层有关的性能问题。Unix 编译器的 -S 选项和 Microsoft 编译器的 /Fa 选项，可以显示编译器所生成的汇编代码。如果使用 GCC 的时候，想查看采用 Intel 语法而非 Unix 语法所写的汇编代码（参见第 37 条），那么可以在调用 gcc 时指定 -masm=intel 选项。对于编译成 JVM 字节码的语言来说，可以在相应的类文件上面调用带有 -c 选项的 javap 指令。尽管汇编代码看上去比较难懂，但你如果试着把这些指令与源代码对应起来，那么很容易就能看出它们的功能，对于通常的调试工作来说，这样就足够了。

现在举一个例子。下面是一段看上去（可能）没有太大问题的 Java 代码，这段代码用来构建一个很长的空字符串。

```
class LongString {
   public static void main(String[] args) {
      String s = "";
      for (int i = 0; i < 100000; i++)
         s += " ";
   }
}
```

然而在笔者的计算机上面，这段程序竟然花了九秒多才执行完。为了查明原因，我们针对编译后的类文件来运行 javap-c LongString 命令，以查看其中所包含的 JVM 指令。该命令会输出程序清单 6.1 这样的内容。

程序清单 6.1　反汇编后的 Java 代码

```
class LongString {
 public static void main(java.lang.String[]);
  Code:
    0: ldc       #2 // String
    2: astore_1
    3: iconst_0
    4: istore_2
    5: iload_2
    6: ldc       #3 // int 100000
    8: if_icmpge 37
   11: new       #4 // class StringBuilder
   14: dup
   15: invokespecial #5 // Method StringBuilder."<init>":()V
   18: aload_1
   // Method StringBuilder.append:(LString;)LStringBuilder;
   19: invokevirtual #6
   22: ldc       #7 // String
   // Method StringBuilder.append:(LString;)LStringBuilder;
   24: invokevirtual #6
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
  // Method StringBuilder.toString:()LString;
  27: invokevirtual #8
  30: astore_1
  31: iinc      2, 1
  34: goto  5
  37: return
}
```

大家可以看到，在 5 ~ 34 号字节所表示的这个循环中，编译器会先创建一个新的 StringBuilder 对象（11 ~ 15 号字节），然后把 s 添加到对象尾部（18 ~ 19 号字节），接下来将”“添加到对象尾部（22 ~ 24 号字节），最后把该对象转换成 String（27 号字节）并把结果存回 s（30 号字节）。由此可见，循环体中的代码，实际上就相当于下面这条开销很大的 Java 语句：

```
s = new StringBuilder().append(s).append(" ").toString();
```

我们可以把 StringBuilder 构造器和字符串转换操作从循环里面拿出来，于是就得到了下面这段效率更高的等价代码，它几乎可以瞬间执行完毕。

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 100000; i++)
   sb.append(" ");
s = sb.toString();
```

要点

- 查看自动生成的代码，以理解源代码中与之对应的编译时和运行时问题。
- 通过编译器的选项或特定的工具，把这些自动生成的代码展示成易于阅读的形式。

第 51 条：使用静态程序分析工具

用工具来替你进行调试，这种想法听上去似乎太过美妙，但实际上，确实是有可能做到的。有很多所谓的静态分析工具，可以在不运行代码的前提下（这也就是“静态”一词的含义），对代码进行扫描，并发现其中较为明显的 bug。其中的某些工具，或许已经成为了基本调试环境中的一部分，因为当前的编译器与解释器，通常本身就能够执行一些简单的静态分析。此外，还有一些独立的工具可供选用，例如，GrammaTech 的

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

CodeSonar、Coverity Code Advisor、FindBugs、Polyspace Bug Finder，以及种种以“lint”结尾的程序。这些分析工具会根据形式化方法（也就是那种基于大量数学运算的算法）以及启发式方法（heuristic，这是个听起来很高端的词，实际上指的就是那种较为合理的猜测而已）来执行操作。按道理来说，我们应该在软件开发的全过程中持续使用静态分析工具，以确保代码干净整洁，如果做不到这一点，也可以单单在调试环节使用这些工具，以查找那些很容易忽视的 bug，诸如与并发有关的问题（参见第 62 条）以及那些有可能导致内存遭到破坏的问题。

某些静态分析工具可以检查出上百种不同的 bug。下面列出其中几种较为重要的 bug，并给出一些简单的范例。在实际工作中，与这些错误有关的代码，通常都会和其他代码纠缠在一起，或是会出现在项目中的很多个例程里面。

- 对 null 进行解引用操作。

```
Order o = null;
if (x)
  o = new Order();
o.methodCall(); // o might still be null
```

- 并发错误与竞争条件（race condition）。

```
class SpinWait {
  private boolean ready;
  public void waitForReady() {
    /*
     * The Java compiler is allowed to hoist the field read
     * out of the loop, making this an infinite loop. Calls
     * to wait and notify should be used instead.
     */
    while (!ready)
      ;
  }
}
```

- 在不需要明确声明变量的编程语言中，使用拼写有误的变量名。

```
if (!colour)
  color = getColor(); // Tested colour, but set color
```

- 用错误的值来充当数组或内存缓冲区的下标（index）。

```
int[] a = new int[10];
for (i = 0; i <= a.length; i++)
      a[i] = 1; // a[a.length] uses an out of bounds index
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 错误的条件语句、循环语句、case 语句，以及永远不会执行到的代码。

```
for (;;)
  n += processRequest();
return n; // This stament will not be executed
```

- 未处理的异常。

```
public void readData(java.io.InputStream is) {
  try {
    is.read();
  } catch (java.io.IOException ex) {
    // The exception is ignored
  }
}
```

- 没有用到的变量和例程。
- 数学错误。

```
int d = 1;
int q = n / --d; // Division by zero
```

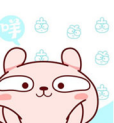
- 代码重复。
- 在实现类的时候，忘记了本来应该编写的那些例行代码（boilerplate），比如，在 C++ 程序中没有遵循 Rule of three（三法则）/Rule of 0（零法则）[⊖]，在 Java 代码中没有同时实现 equals 方法及 hashCode 方法等。
- 资源泄漏。

```
void writeDone()
{
   FILE *f = fopen("myfile", "w");
   fprintf(f, "Done\n");
   return;
   // The opened file stream cannot be closed
}
```

- 安全漏洞。

```
/*
 * Input longer than the buffer size will overflow the
 * buffer allowing a stack smashing attack.
 */
gets(buff);
```

⊖ 也就是说，要么同时实现析构函数、复制构造函数和赋值运算符这三者（三法则），要么任何一个都不要实现（零法则，也称为 Rule of Zero）。——译者注



非卖品，仅供非商业用途或交流学习使用

- 与特定的编程语言有关的问题。

```
public static void main(String[] args) {
  // This compares objects rather than strings
  if (args[0] == "--help")
```

要注意的是，代码分析工具可能会错过人类能够发现的某些 bug（这叫做假阴性，false negative），也有可能会针对正确的代码发出警告（这叫做假阳性，false positive）。比如，4.8.3 版本的 GCC，就找不出下面这个函数中的问题。该函数的问题在于：程序运行到函数尾部时，变量 r 可能仍然没有正确地初始化。

```
int
myFunction(int c)
{
      int r;

      if (c)
            r = 0;
      return r;
}
```

与之形成对照的是，Visual Studio 2015 的 C 语言编译器，有时会针对正确的代码给出警告。如下面这个函数。当程序运行到函数尾部时，变量 r 应该已经得到了正确的初始化，但是编译器依然会给出警告，说该变量有可能没有初始化。

```
int
myFunction(int c)
{
      int r, c2 = 0;

      if (c)
            r = 9;
      else
            c2 = 1;
      if (c2)
            r = 0;
      return r;
}
```

没有哪一种静态分析工具是完美的，因为它们都要受到实际因素和理论因素这两个方面的限制。从前一方面来说，由于执行路径的数量会呈指数式增长，因此，要想把程序可能具备的所有状态全都探索到，需要相当大的内存。从后一方面来说，由于某些底层问题本身就是不可判定的，因此无论采用什么样的算法都不能够正确地求解。由此可见，尽管静态分析工具对调试工作很有帮助，但有的时候，你必须自己来判断这些工具

非卖品，仅供非商业用途或交流学习使用

所给出的结果，并且要注意它们所无法探查到的那些情况。

要想体会到静态分析工具所带来的好处，你首先应该关注自己当前正在使用的这款编译器或解释器。它们或许会提供一些选项，用来对代码执行更为严格的检查，并且会在发现可疑代码时给出警告。例如：

- 对于 GCC、ghc（Glasgow Haskell Compiler）及 clang（为 LLVM 编译器而开发的前端程序，支持 C 系列的编程语言）来说，可以从 -Wall、-Wextra 及 -Wshadow 这样的选项入手（此外还有很多类似的选项可供选用）。
- 对于 Microsoft 的 C/C++ 编译器来说，对应的选项是 /Wall 和 /W4。
- 在 JavaScript 程序中，可以通过 "use stric"; 指令来开启严格模式（就是带双引号的）。
- 在 Perl 程序中，可以使用 use strict; 和 use warnings; 指令来进行更为严格的检查。（Perl 和 JavaScript 的那些选项，会同时执行静态检查与动态检查。）使用编译器时，也可以指定一种较高的优化级别，令其能够进行某些特定的分析，这些分析，有助于编译器针对源代码给出一些警告。如果警告的级别可以调整，那么应该尽量调高，只是要注意，不要过多地生成一些与待解决的问题无关的警告信息。看到这些警告信息之后，就应该有条理地去解决它们。这样做，或许能够解决你当前正在调试的这个问题，或是可以令你更为容易地看到将来有可能出现的问题。

把警告消除干净之后，可以趁势调整编译器的选项，令其将所有的警告都视为错误（Microsoft 的编译器使用 /WX 选项，GCC 使用 -Werror 选项）。开启该选项之后，警告信息就不会淹没在冗长的输出文字中了，而是会直接令编译器报错，从而迫使开发者必须写出绝对不会引发警告的代码。

把编译器配置好，可以令你享受到它所带来的很多好处，然而在这个基础上，还可以考虑再配置一些分析工具。这些工具可能要花更长的时间来检测 bug，而且有可能出现更多的误报。维基百科的 static analysis tools 页面列出了上百种工具，其中既有 Coverity Code Advisor 这样的流行商业产品，又有 FindBugs 这样的知名开源软件。某些工具专注于寻找特定类型的 bug，如安全漏洞或并发问题。你要选择符合自己需求的工具，而且有的时候，应该把几种互补的工具结合起来使用，以便发现更多的 bug。此外，还应该花些功夫把每一个工具都配置好，将那些与自己的编程风格不符的警告选项关掉，以尽量减少误报的情况。

最后，要把静态分析环节，配置成构建软件系统时所应执行的一个步骤，并且令其成为持续集成的一个部分。通过调整软件项目的构建配置，开发者可以对检测代码所用的静态分析工具进行统一的管理，而且这些工具也有可能会在持续集成的过程中，找到开发者原来没有发现的一些问题，并立刻给出汇报。这样做使我们总是能够保证：软件项目里面不会含有静态分析工具所能检查到的错误。要注意防止这样一种情形：项目团队（可能为了寻找某个难于捕获的 bug，从而）决心花很大的力气来扫除这些由静态分析工具所指出的错误，但是却没有能够坚持把它们清除干净，于是，又会有新的错误在代码中滋生。

要点

- 某些专用的静态程序分析工具，有可能会找到很多潜在的 bug，其数量要比编译器所能给出的警告更多。
- 把编译器配置好，令其能够对程序进行适当的分析，并找出其中的 bug。
- 至少要将一款静态程序分析工具，纳入你的构建流程和持续集成流程中。

第 52 条：对项目进行配置，令程序能够以固定的方式构建和执行

下面这个程序，会打印出与自身的栈、堆、代码及数据有关的内存地址。

```
#include <stdio.h>
#include <stdlib.h>

int z;
int i = 1;
const int c = 1;

int
main(int argc, char *arg[])
{
      printf("stack:\t%p\n", (void *)&argc);
      printf("heap:\t%p\n", malloc(1));
      printf("code:\t%p\n", (void *)main);
      printf("data:\t%p (initialized)\n", (void *)&i);
      printf("data:\t%p (constants)\n", (void *)&c);
      printf("data:\t%p (zero)\n", (void *)&z);
      return 0;
}
```

在许多开发环境下，每次运行这个程序所得到的结果都各不相同。（笔者在 GNU/Linux 系统的 GCC，OS X 系统的 clang 以及 Windows 系统的 Visual C 下面试过了，都是如此。）

```
$ dbuild
stack: 003AFDF4
heap: 004C2200
code: 00CB1000
data: 00CBB000 (initialized)
data: 00CB8140 (constants)
data: 00CBCAC0 (zero)
$ dbuild
stack: 0028FC68
heap: 00302200
code: 01331000
data: 0133B000 (initialized)
data: 01338140 (constants)
data: 0133CAC0 (zero)
```

之所以会出现这种现象，是因为操作系统的内核每次都会把程序加载到内存中某个随机的地址上面，以防止那种针对程序码所发起的恶意攻击。这种代码注入攻击，是想用恶意代码来填满程序的缓冲区，并使其溢出，然后诱导这个程序去执行溢出后的那些代码。如果程序中的各个部分每次都出现在内存里的固定位置上面，那么这样的攻击就很容易实施。因此，为了进行反制，某些内核会随机安排程序的内存布局，以防止这种通过硬编码的内存地址来进行的注入式攻击。

然而问题在于，这种地址空间布局随机化（address space layout randomization，ASLR）措施，也有可能对调试工作起到干扰作用。例如，它会使我们在某一次运行程序时所看到的故障，不会发生在下一次的运行过程中；会使我们费心记录下来的指针值，在下一次运行程序的时候发生改变；会使基于内存地址所构造出来的哈希表，在下一次运行程序时以其他方式得到填充；也会使某些内存管理器的行为，在每次运行程序时都有所区别。

因此，在调试程序时，我们有可能想要使程序的行为能够在历次的执行过程中保持稳定，如果正在调试的是与内存有关的问题，那就更应该如此。在 GNU/Linux 系统中，可以采用下面这种方式来运行程序，以禁用 ASLR：

```
setarch $(uname -m) -R myprogram
```

在 Visual Studio 平台中，你可以在链接代码时指定 /DYNAMICBASE:NO 选

项，以禁用 ASLR 功能，也可以通过设置开发项目的 Configuration Properties-Linker-Advanced-Randomized Base Address 选项来实现。某些版本的 Windows 系统有一个注册表项，可以在整个系统里面禁用 ASLR（HKLM/SYSTEM/CurrentControlSet/Control/SessionManager/MemoryManagement/MoveImages）。在 OS X 系统上面，你需要通过编译器的 -Wl 标志来把 -no_pie 选项传给链接器，也就是说，你需要按照下面这种方式来进行编译：

```
-Wl,-no_pie -o myprogram myprogram.c
```

在用同一份源代码来构建应用程序时，还有一些因素也会使得每次构建出来的程序有所不同，所幸其幅度没有内存地址方面的区别那么大。下面列出一些有代表性的地方：

- GCC 会随机选取一个独特的符号名称，并将其放在每一份编译好的文件中。你可以通过 -frandom-seed 标志，把这些名称固定下来。
- 输入给编译器的文件顺序可能有所不同。如果有待编译或有待链接的文件，是通过 Makefile 的通配符扩展机制来指定的，那么它们之间的顺序，就会有可能因为目录中的项目发生重排而得以改变。若想把顺序固定下来，你可以明确地列出这些文件，也可以对扩展后的文件进行排序。
- 在软件中用来表示版本的时间戳，会随着构建软件的时间而发生变化，如 __DATE__ 和 __TIME__ 这两个宏就是如此。如果你想把它们固定下来，那么可以改用版本控制系统的版本标识符（如 Git 的 SHA 校验和），以便能够在有需要的时候，根据该标识符来构造时间戳。
- 对哈希表或图（map）进行遍历后所得到的列表，其元素顺序可能会有所不同。某些编程语言为了防止算法复杂度攻击，每次都会采用不同的方式来为对象进行哈希，这会导致每次遍历容器时所看到的元素顺序有所区别。要想解决这个问题，我们可以对遍历之后所产生的列表进行排序。对于 Perl 和 Python 语言来说，也可以分别通过设置 PERL_HASH_SEED 及 PYTHONHASHSEED 环境变量来解决。
- 加密盐（encryption salt，为了加密而引入的干扰因素）。加密程序通常会采用随机得到的某个值（也就是盐）来对提供给它的数据进行扰乱，从而防止那种通过预先构造字典所实施的攻击（pre-built dictionary attack）。在测试和调试程序的时

候，可以关掉这种扰乱机制。比如，openssl 程序就提供了 -nosalt 选项。然而要注意，发布到生产环境中的程序，不应该使用这个选项，因为那样做会令系统易于遭受字典式的攻击。

从镜像文件的一致性方面来看，构建程序的最高标准，是能够在不同的主机上编译同一份源代码，并且得到一模一样的软件包。为此，我们需要执行大量的工作，因为它还涉及文件路径、语言配置、档案的元数据、环境变量以及时区等诸多因素。如果你确实需要实现到这种程度，那么可以参考 reproducible builds 网站，该网站提供了很多可靠的建议，告诉你应该怎样解决此类问题。

要点

- 对软件的构建流程和执行过程进行配置，使其运行效果能够重现。

第 53 条：对调试所用程序库及构建代码时所应执行的检查进行配置

在对代码和程序库进行编译与链接时，有很多选项都能对程序在运行时所执行的操作，做出更为严格的检查。这些选项，与我们在配置软件的调试模式时所使用的那些选项（参见第 40 条），是可以协同运作的，因此，我们在编译期应该同时启用这两类选项。本条目所讲的选项，主要适用于 C、C++ 和 Objective-C 语言，而这三种语言为了提升执行效率，在一般情况下，不会执行缓冲区越界检查，因此，如果开启了这些检查，那么程序可能会运行得较为缓慢。在实时系统或对性能要求很高的环境中使用这些方法时，要多加注意。下面大家会看到一些常见的配置方法，在编译和链接代码的过程中，可以通过这些方法来找寻与内存的使用情况有关的 bug。

采用 C++ 标准模板库（STL）来编程时，可以对程序代码执行很多种检查。如果使用的是 GNU 版本，那就在编译代码时，定义名为 _GLIBCXX_DEBUG 的宏，如果是在 Visual Studio 环境下编程，那就以 debug 模式来构建项目，或把 /MDd 选项传给编译器。启用了这些与 STL 有关的检查之后，我们可以在构建过程中发现代码里面的一些问题，例如，对迭代器进行越界的递增操作、对已销毁的容器所对应的迭代器进行解引用，或是违反算法的前置条件等。下面举一个例子：

```
#include <vector>

int
main()
{
      std::vector<int> v;
      v[0] = 3;
}
```

如果用 GCC 编译该程序，那么它在运行的时候就会给出内容为“attempt to subscript container with out-of-bounds index 0, but container only holds 0 elements”的错误消息，如果用 Visual Studio 编译它，那么运行时的错误消息则是“vector subscript out of range”。再来看一个例子。在下面这个程序中，用来求交集的 s1 和 s2，本来应该是已经排好顺序的 vector：

```
#include <vector>
#include <algorithm>
#include <iterator>

int
main()
{
    std::vector<int> s1 = {5, 3, 2};
    std::vector<int> s2 = {1, 3, 2};
    std::vector<int> result;

    std::set_intersection(s1.begin(), s1.end(),
                          s2.begin(), s2.end(),
                          std::back_inserter(result));
}
```

然而该程序并没有提前把 s1 和 s2 排好顺序，因此，如果编译的时候开启了 STL 检查选项，那么程序在运行时，就会显示出内容为“elements in iterator range[__first1, __last1)are not sorted”（适用于 GCC）或“sequence not sorted”（适用于 Visual Studio）的消息。此外，还可以在编译的时候定义一个名为 _GLIBCXX_DEBUG_PEDANTIC 的宏，以便把代码中所用到的某些特性指出来，那些特性无法移植到其他的 STL 实现版本上面。

GNU 的 C 语言程序库也可以对内存泄漏问题进行检查，看看程序在整个生命期里面，有没有把分配过的内存全都释放掉。要想执行该检查，我们需要在程序开头调用 mtrace 函数，然后在运行程序的时候，设置名为 MALLOC_TRACE 的环境变量，并将变量值设为一个文件，用来保存这项检查所输出的信息。下面这个程序，在其退出的时

候，并没有把自己所分配的内存块释放掉：

```
#include <stdlib.h>
#include <mcheck.h>

int
main()
{
#ifndef NDEBUG
      mtrace();
#endif
      char *c = malloc(42);
      return 0;
}
```

针对程序所生成的输出文件来运行 mtrace 命令，就可以准确地看到泄漏的内存块是在什么地方分配的。

```
Memory not freed:
-----------------
  Address  Size   Caller
0x090e5378 0x2a at leak.c:10
```

还有一个更为通用（而且开销也更大）的办法，可以检测 C 与 C++ 代码在内存访问方面的各种问题，这就是 AddressSanitizer（ASan）。GNU 编译器和 LLVM Clang，都可以通过 -fsanitize=address 选项来启用这个检测系统。为了获得更为清晰的结果，还可以同时开启 -g 与 -fno-omit-frame-pointer 选项。下面这个程序会越过数组的边界，因为它误把 sizeof(a) 的结果当成了数组的长度（实际上，还应该把这个结果与元素的大小相除）：

```
int
main()
{
      int i, a[5];

      for (i = 0; i < sizeof(a); i++)
            a[i] = i;
}
```

编译上述代码的时候，启用 AddressSanitizer 机制，然后运行程序，就可以看到下面这样的错误消息：

```
==59468==ERROR: AddressSanitizer: global-buffer-overflow on
address 0x000100615134 at pc 0x000100614eb0
WRITE of size 4 at 0x000100615134 thread T0
```

非卖品，仅供非商业用途或交流学习使用

```
    #0 0x100614eaf in main oob.c:7
    #1 0x7fff926f15ac in start
    #2 0x0 (<unknown module>)

0x000100615134 is located 0 bytes to the right of global
variable 'a' defined in 'oob.c:4:16' (0x100615120) of
size 20
SUMMARY: AddressSanitizer: global-buffer-overflow oob.c:7 main
```

采用某些编译器进行编译时，可能需要再传入一些信息，使得错误报告中所列出的是源代码的位置，而非机器码的内存地址。比如，你可能要把 ASAN_SYMBOLIZER_PATH 这个环境变量，指向某个能够进行此种映射的程序（如 /usr/bin/llvm-symbolizer-3.4），并且要把 ASAN_OPTIONS 环境变量，设为 symbolize=1。

很多系统都支持 AddressSanitizer 功能，其中包括运行在 i386 和 x86_64 CPU 上面的 GNU/Linux、OS X 以及 FreeBSD，也包括运行在 ARM 架构上的 Android，还包括 iOS Simulator。AddressSanitizer 会急剧增加程序的开销，使其对内存和处理器的占用量，变为平常的两倍左右。但这样做也是有好处的：由于该机制很少会发生误报，因此在调试软件的过程中，可以用它来可靠地搜寻并移除很多与内存有关的问题。

Visual Studio 对内存分配和内存访问方面的错误也可以做出检查，虽然这套机制并没有 AddressSanitizer 那样强大，但是依然能够应对很多情况。要想启用这套机制，我们需要在与 C 语言调试库进行链接的时候，传入 /MDd 选项，而且要像下面这个程序一样，定义一个宏，并调用几个适当的函数：

```
// Define in order to get file and line number information
#define _CRTDBG_MAP_ALLOC

#include <stdlib.h>
#include <crtdbg.h>

int
main()
{
   // Send output to stderr, rather than the VS debug window
   _CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_WARN, _CRTDBG_FILE_STDERR);

   // Detect memory leaks on exit
   _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
   {
      char *c = malloc(42);
      c[42] = 'a';
```

非卖品，仅供非商业用途或交流学习使用

```
    }
    // Check all blocks for memory buffer overflows
    _CrtCheckMemory( );

    return 0;
}
```

运行程序之后，会看到下面这样一份错误报告，其中指出了程序所遭遇的堆受损及内存泄漏问题。

```
HEAP CORRUPTION DETECTED: after Normal block (#106) at
0x00688EF0. CRT detected that the application wrote to
memory after end of heap buffer.

Memory allocated at memerror.c(19).
Normal located at 0x00688EF0 is 42 bytes long.

Memory allocated at memerror.c(19).
Detected memory leaks!
Dumping objects ->
memerror.c(19) : {106} normal block at 0x00688EF0,
42 bytes long.
```

请注意，这套机制只能识别出那种在分配好的堆之外所发生的写入行为，而不能像 AddressSanitizer 那样，识别出无效的读取操作，以及对全局内存和栈内存的无效访问操作。

在 OS X 系统下，以及开发 iOS 应用程序的时候，还有一种办法也可以做内存检测，那就是与 Guard Malloc（libgmalloc）程序库相链接。这样做会把每一个分配好的内存块，都放在单独的（也就是不连续的）虚拟内存页面中，使得系统能够探测到页面之外所发生的内存访问行为。该方式在分配内存时，会使虚拟内存系统的负载加大，但是在对内存访问行为进行检查时，则不需要占用额外的 CPU 资源。这种方式适用于 C、C++ 和 Objective-C 语言。为了使用这个程序库，我们需要把环境变量 DYLD_INSERT_LIBRARIES 的值，设为 /usr/lib/libgmalloc.dylib。还有一些环境变量也可以用来对该程序库的功能进行微调，详细情况请参见 libgmalloc 的帮助手册页面。

例如，如果在构建下面这个程序的过程中，能够与 libgmalloc 库相连接，那么在运行程序时，系统就会提示内存区块错误（segmentation fault）并终止该程序。

```
int
main()
{
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
    int *a = new int [5];
    int t = a[10];
    return 0;
}
```

得知这个错误之后，我们很容易就能通过调试器来精确地找到与该错误有关的代码。

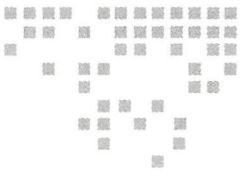
如果上面讲的那些机制，全都无法在你所处的环境中使用，那么应该考虑用一种可以进行检查的程序库来替换当前所使用的程序库。dmalloc 就是个很好的例子，这是一个带有调试功能的程序库，可以方便地替换 C 语言默认的内存分配函数。

要点

- 在你所处的开发环境中，找出编译器和程序库所支持的运行时调试功能，并启用这些功能。
- 如果开发环境里面没有这样的功能，那么可以考虑在构建软件的过程中，纳入一些包含该功能的第三方程序库。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



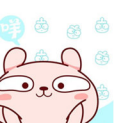
第 7 章 Chapter 7

运行时的调试技术

要看程序有没有 bug，最终还是得通过执行来做出判断。把程序运行起来之后，各个方面的情况就可以暴露出来了，例如，运行得是否正确、对 CPU 和内存的占用量是否合理等，此外，我们还可以看出它与（可能包含 bug 的）程序库、操作系统及硬件之间的交互情况。但是，由于计算机每秒钟会执行数十亿条指令，因此，这些情况可以说是转瞬即逝的。如果你想把它们捕获，那将是一件很棘手、很麻烦的事情，有时甚至完全无法可靠地进行捕获。于是，我们应该利用测试、应用程序日志以及监测工具来审视程序在运行时的行为，以便将那个困扰我们的 bug 找出来。

第 54 条：通过构建测试用例来寻找错误

通过适当的测试，我们通常能够找到乃至修复某个 bug。有些人把这种方法称为“Defect-Driven Testing”（缺陷驱动测试），它的首字母缩写恰好与知名的杀虫剂一样，都叫 DDT。下面给出这种测试方法的三个步骤，并举一个实际的例子。这个例子是根据 qmcalc 程序中的真实 bug 改编而来的。qmcalc 程序能够根据各种指标来计算 C 语言代码文件的编写质量并给出评分，然后把这些评分放在一个以制表符（tab）来分隔的列表中。问题是：程序本来应该输出 110 个字段，但是在极个别的情况下，其所输出的字段数量却比较少。



非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

首先，创建测试用例，以便可靠地重现你想要解决的问题。这意味着我们需要指定程序出错所经的流程，以及展现该错误所需的材料（通常是数据）。比如，某个测试用例可能会把 foo 文件（这是材料）加载进来，然后依次按下 x、y、z 按钮，使应用程序崩溃（这是流程）。又如，某个测试用例会把 Acme 的负载均衡器（这是材料）运用到应用程序上面，从而导致用户刚开始使用程序时所需完成的认证环节出错（这是流程）。

在接下来要讲的这个例子中，我们通过下列命令来针对 Linux 的所有 C 语言文件运行 qmcalc 程序，并统计该程序为每个文件所生成的字段数量。

```
# Find all C files
find linux-4.4 -name \*.c |
# Apply qmcalc on each file
xargs qmcalc |
# Display the number of fields
awk '{print NF}' |
# Order by number of fields
sort |
# Display number of occurrences
uniq -c
```

通过上述结果，我们可以看到，这个程序在个别情况下，所生成的字段数量是少于 110 个的。

```
    8 100
   19 105
21772 110
   12 80
  472 90
```

第二步是简化测试用例，令其刚好能够重现你想要解决的问题（参见第 10 条）。有两种简化的办法，第一种是从头开始构建测试用例，直到 bug 突然出现，第二种是逐渐缩减现有的测试用例，直到 bug 突然消失。此时，测试用例中所含的数据，通常就能够帮你找到问题了，甚至还能够直接给你指出解决办法。在很多情况下，这两种方式都可以结合起来使用，也就是先尽可能多地删减测试代码，等你觉得自己已经找到问题之后，再重新开始，构建一个最简的测试用例。

针对同样一个问题，我们采用下面这些 shell 命令把 qmcalc 程序所无法正确处理的首个文件显示出来：

```
# Find C files
find linux-4.4/ -name \*.c |
# For each file
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
while read f ; do
  # If the number of fields is not 110
  if [ $(qmcalc $f | awk '{print $NF}') != 110 ] ; then
    echo $f  # Output the file name
    break    # Stop processing
  fi
done
```

运行上述管道程序之后，我们会发现，linux-4.4/drivers/mmc/host/sdhci-pci-o2micro.c 文件最先导致 qmcalc 程序所输出的结果少于 110 个字段。

如果只从该文件里取出一部分内容给 qmcalc 处理，那还会不会出现同样的问题呢？下面这几条命令，会逐渐缩减待处理的行数，并显示出由 qmcalc 程序所生成的字段数量。

```
$ cp linux-4.4/drivers/mmc/host/sdhci-pci-o2micro.c test.c
$ ./qmcalc test.c | awk '{print NF}'
87
$ head -100 test.c | ./qmcalc | awk '{print NF}'
87
$ head -10 test.c | ./qmcalc | awk '{print NF}'
63
$ head -1 test.c | ./qmcalc | awk '{print NF}'
63
```

接下来，手工构建一个测试，把 qmcalc 程序运用到 /dev/null 这个空文件上面，以便继续简化测试用例。

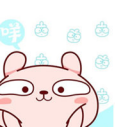
```
$ ./qmcalc /dev/null | awk '{print NF}'
59
```

我们以一种清晰的形式来查看程序的输出结果，这样立刻就会发现其中有一些连续出现的制表符，而这种现象，很可能说明程序的错误与一组空字段有关。

```
$ ./qmcalc /dev/null | sed -n l
0\t0\t\t\t\t0\t0\t\t\t\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t\t\
\t\t0\t\t\t\t0\t\t\t\t0\t\t\t\t0\t\t\t\t0\t0\t0\t0\t0\t0\t0\t0\
\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t\t\
0\t0\t0\t0\t0\t0\t0\t0\t0\t0\t0$
```

于是，只要查看源代码，就可以发现问题的原因：原来程序在输出描述性的统计结果时，并没有能够正确地处理空集。它本来应该输出四个字段分隔符（也就是制表符），但实际上只输出了三个。

```
template <typename T>
std::ostream&
```



非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
operator <<(std::ostream& o, const Descriptive<T> &d) {
   if (d.get_count() != 0)
      o << d.get_count() << '\t' << d.get_min() << '\t' <<
         d.get_mean() << '\t' << d.get_max() << '\t' <<
         d.get_standard_deviation();
   else
      o << "0\t\t\t";
   return o;
}
```

第三步，是巩固胜利成果。把程序中的问题隔离出来之后，要趁机给代码添加对应的单元测试或回归测试（参见第 42 条）。如果问题与隔离出来的这些代码里面的某个错误有关，那就应该添加对应的单元测试，如果问题是由多种因素合起来导致的，那么回归测试或许更为合适。这种回归测试，应该对测试用例进行打包，使得它们能够在例行的软件测试过程中，得以自动执行。当错误依然位于代码中时，我们要运行软件测试，以确保该测试是无法通过的，这样才能说明它确实捕获到了所要解决的问题；而当我们把错误修改好之后，该测试应该就可以正确地运行了，我们由此也可以确信，自己确实解决了这个问题。此外，这种测试还能够确保程序在以后不会出现同样的问题。

笔者针对刚才所说的那个问题，给 qmcalc 项目中添加了下面这样一个 CppUnit 测试用例：

```
void testOutputEmpty() {
   std::stringstream str;
   Descriptive<int> a;
   str << a;
   CPPUNIT_ASSERT(str.str() == "0\t\t\t\t");
}
```

在没有修复错误代码之前，这个单元测试应该是无法运行通过的。

```
$ make test
./UnitTests
.............................................................
..........F..................................
!!!FAILURES!!!
Test Results:
Run: 110 Failures: 1 Errors: 0

1) test: DescriptiveTest::testOutputEmpty (F) line: 103
DescriptiveTest.h assertion failed
- Expression: str.str() == "0\t\t\t\t"
```

把漏掉的那个字段分隔符添上之后，代码中的错误就修复好了，而这个测试用例，

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

也能够正确地运行了。

```
$ make test
.....................................................................
.....................................................

OK (110 tests)
```

用 Andrew Hunt 和 David Thomas 的话来说就是：“把所有的测试都通过之后，代码才算写完。”

针对已经解决的问题来添加测试用例，或许并不是多此一举的事情。首先，你可能会忘记修复某种特定的情况，而这个测试正好能够在执行代码的时候帮你找到那种情况。其次，如果有人对合并冲突没有做出正确的处理，那么可能会再度引入同样的错误。第三，以后也可能有人会出现类似的错误。第四，这种测试或许还能够捕获到与之相关的其他错误。因此，编写测试的时候，真的不能太过简省了。

用测试来探查 bug 时，我们应该要了解：代码中的哪些部分是实际上能够测试到的，而哪些部分又是不太能够为测试所覆盖的。之所以要了解这一情况，是因为在受测程度较低的那一部分代码中，有可能隐藏着 bug。我们可以使用一种能够执行测试覆盖率分析的工具来掌握代码的受测情况。这种工具包括：适用于 C 和 C++ 的 gcov（参见第 57 条）、适用于 Java 的 JCov、JaCoCo 和 Clover、适用于 .NET 的 NCover 和 OpenCover、适用于 Python 的 coverage 包，以及适用于 JavaScript 的 blanket.js。

要点

- 创建一个可靠且最简的测试用例，在这个过程中，你有可能会发现程序中的问题及其解决办法。
- 把测试用例作为单元测试或回归测试，嵌入软件中。

第 55 条：令软件在遇到问题时尽早退出

迅速而高效地重现问题，有助于提升调试的效率（参见第 11 条和第 10 条）。因此，一旦发现软件有出错的迹象，就应该立刻令其停止运行。这样做可以使你更为容易地找到与之相对应的错误，因为停止软件运行所用的这部分代码，就是在执行完错误的代码之后才得以执行的，有的时候，这两者之间的距离还会相当接近。反之，如果在软件遇

非卖品，仅供非商业用途或交流学习使用

到小问题的时候，听任其继续执行，那就会使软件进入未知的状态中，进而引发一系列其他的问题，这导致我们很难找到最初的那个 bug。

使软件尽早暴露出故障，有可能会把你引向与待解决的这个问题无关的其他问题上面。然而，你依然可以先把那个问题修复了，然后重新启动调试流程，这样做至少能够永久地排除一项疑点。如果能够逐渐地消除这些问题，那么你就会在调试的过程中持续取得进展。反之，若是一直把这些小问题拖着不管，则迟早会酿成大错。

下面这些办法可以使程序尽快暴露出其中的错误：

- 添加并启用断言语句，以验证输入给例程的参数以及对 API 的调用是否正确（参见第 43 条）。对于 Java 语言来说，可以在运行的时候通过 -ea 选项来启用断言。对于 C 和 C++ 语言来说，通常可以在编译的时候，取消对 NDEBUG 宏标识符的定义，以启用断言功能。（在构建 production 版本的程序时，该标识符通常是已定义好的。）
- 对程序库进行配置，令其对自身的用法执行严格的检查（参见第 53 条）。
- 用动态程序分析工具来检查程序所执行的操作（参见第 59 条）。
- 启动 Unix shell 的时候，开启 -e 选项，使得该 shell 能够在脚本命令发生错误时（也就是其退出状态码非 0 的时候）终止。

请注意，对于自成一体的程序来说，令其在遇到错误时尽快退出，是一种可以提高调试效率的做法，然而，这对于那种已经从开发环境搬移到维护环境中的大型生产系统来说，可能就不太合适了。在那种情况下，最先保证的应该是程序的恢复能力，也就是说，当程序发生小的错误（例如，加载图标时出现问题，或是多个服务器进程中的某一个发生崩溃）时，我们通常应该容许它继续执行，而不是令整个系统都停止运行。我们可以通过其他措施来与这种较为宽松的操作模式相平衡，如进行广泛的监测（参见第 27 条）及日志记录（参见第 56 条和第 41 条）等。

要点

- 调试程序的时候应该设置一些机关，使得程序能够在刚刚有出错的迹象时，就退出。

第 56 条：检视应用程序的日志文件

有很多程序要在后台执行复杂的处理，并且没有提供通过控制台来进行访问的机

非卖品，仅供非商业用途或交流学习使用

制，这些程序会把它们所执行的操作，写到某个文件或某种特殊的日志收集设施中。这些日志文件里面的内容，使我们能够了解该程序的实际执行情况，并且能够根据自己的需要来对一系列的事件进行分析。当程序出现故障之后，我们有可能会在日志中发现一条错误消息或警告消息（例如，“Unable to connect to example.com:Connection refused”（无法连接到 example.com，连接遭到拒绝）），从而了解故障背后的原因，也有可能会在日志中发现某份数据，从而找到软件或配置中的错误。因此，当我们着手调查软件故障时，总是应该先从日志文件入手，这是个良好的习惯。

不同的操作系统和软件平台，会采用各种存储方式来把日志文件保存到不同地方。Unix 系统的日志文件，通常以文本形式保存到 /var/log 目录中。应用程序可以在该目录创建它们自己的日志文件，也可以把事件记录到与该事件类型相符的现有文件中。下面举一些例子：

- 身份认证：auth.log
- 后台进程：daemon.log
- 内核：kern.log
- 调试信息：debug
- 其他信息：messages

在不太繁忙的系统中，我们或许可以通过下列命令来寻找与正在调试的这款应有程序相对应的那份日志文件：

```
ls -tl /var/log | head
```

上面那条命令，可以列出最近有所改动的文件，如果应用程序刚刚在某份日志文件里面写入了一条记录，那么那份文件的名称，就会出现在文件清单的开头。此外，如果要找的文件位于 /var/log 的子目录中，那么可以通过下列命令来查找：

```
# List all files under /var/log
find /var/log -type f |
# List each file's last modification time and name
xargs stat -c '%y %n' |
# Order by time
sort -r |
# List by the ten most recently modified files
head
```

如果这些办法都不行，那么你可以试着在应用程序的文档中查询日志文件的名称，也可以试着在其执行记录（参见第 58 条）或源代码中寻找该文件的名字。

Windows 系统的应用程序日志，是以一种不透明的格式来存储的。可以运行 Eventvwr.msc 命令，以启动名为 Event Viewer 的 GUI 应用程序，并在其中浏览及过滤日志文件，也可以使用 Windows PowerShell 的 GetEventLog 命令，或是对应的 .NET API 来查看。日志分成很多个种类，你可以在 Event Viewer 左侧的树状列表里面，按照类别来进行浏览。在 OS X 系统上面，带有图形界面的日志查看程序叫做 Console。Windows 与 OS X 的日志查看程序，都可以过滤日志的内容、创建自定义的视图，或搜寻特定的条目。在 Unix 系统上面，你可以通过命令行工具（参见第 22 条）来进行类似的处理。

很多应用程序都可以对其所记录的信息量（也就是日志的详细程度，log verbosity）进行调整，我们可以通过命令行选项或配置选项来调整，有时甚至还可以给正在运行的程序发送适当的信号，以调整其 log 数量。此外，日志框架可能还会提供其他一些对日志信息进行扩充或调节的机制。把问题调试好之后，别忘了将日志的详细程度恢复到原来的级别。如果详细程序调得过高，那么程序的性能就有可能下降，而且其占用的磁盘空间或网络带宽也有可能变得过多。

在 Unix 系统中，应用程序会在每条日志消息上面，标出该消息所涉及的方面（如 authorization（身份验证）、kernel（内核）、mail（邮件）、user（用户）等）以及消息的级别（如 emergency（紧急）、alert（警告）、informational（信息）、debug（调试））。有一个运行在后台的程序叫做 syslogd（或 rsyslogd），它负责监听系统中的日志消息，并将其记入文件，我们可以对该程序进行配置，告诉它应该怎样处理特定的消息。与此相关的配置文件叫做 /etc/syslog.conf（或者叫做 /etc/rsyslog.conf，此外，/etc/rsyslog.d 目录里面可能还有一些配置文件）。我们可以命令 syslogd 程序只将那些达到一定级别（例如，只针对 informational 及以上级别的消息，而不针对 debug 级别的消息），并且与特定方面有关的消息记录到文件、发送到控制台或将其忽略。比如，下面这几行配置，可以令 syslogd 程序把 security 方面的全部消息、达到 informational 级别的 authorization 消息，与级别恰好为 debug 的所有消息，分别记录到对应的日志文件中。此外，它还命令 syslogd 程序把级别为 emergency 的消息，发送给所有已经登录的用户。

```
security.*   /var/log/security
auth.info    /var/log/auth.log
*.=debug     /var/log/debug.log
*.emerg      *
```

对于 JVM 代码来说，我们可以采用流行的 Apache log4j 日志框架来更为详细地指

定所要记录的信息内容及其位置。该框架的结构，是基于 logger、appender 及 layout 而搭建起来的，其中，logger 是输出日志消息时所经过的渠道，appender 是一种发送日志消息的机制，可以把消息发送给文件或网络套接字这样的接收方（sink），而 layout 则用来指定每条日志消息的格式。Log4j 是通过文件来配置的，该文件可以按照 XML、JSON、YAML 或 Java 属性文件的格式来书写。下面是一份 log4j 配置文件中的一小段内容，这份文件是提供给 Rundeck 这个工作流与配置管理系统所使用的。

```
# This logger covers all of Grails' internals
# Enable to see whats going on underneath.
log4j.logger.org.codehaus.groovy.grails=warn,\
  stdout,server-logger
log4j.additivity.org.codehaus.groovy.grails=false

# server-logger - DailyRollingFileAppender
# Captures all output from the rundeckd server.
log4j.appender.server-logger=org.apache.log4j.\
  DailyRollingFileAppender
log4j.appender.server-logger.file=/var/log/rundeck/rundeck.log
log4j.appender.server-logger.datePattern='.'yyyy-MM-dd
log4j.appender.server-logger.append=true
log4j.appender.server-logger.layout=org.apache.log4j.\
  PatternLayout
log4j.appender.server-logger.layout.\
  ConversionPattern=%d{ISO8601}[%t] %-5p %c - %m%n
```

对程序所要输出的日志消息进行级别调整，通常有助于我们找到与待调试的问题有关的数据。下面举例来演示怎样通过调整消息的级别来探查 ssh 连接失败的原因。sshd 程序所用的配置文件，叫做 /etc/ssh/sshd_config，该文件里面有一个已经注释掉了的文本行，这行文字用来指定 sshd 程序所用的日志级别（其默认值是 INFO）。

```
#LogLevel INFO
```

当日志级别设为 INFO 时，如果连接失败，那么我们只会看到下面这样的消息。

```
Jul 30 12:49:49 prod sshd[5369]: Connection closed by
10.212.204.48 [preauth]
```

现在我们把日志消息的级别调整为 DEBUG。

```
LogLevel DEBUG
```

这样就可以看到很多条含有一定信息量的消息了，在这些消息中，下面这条消息，清楚地指出了问题的原因：

```
Jul 30 12:57:07 prod sshd[5713]: debug1: Could not open
authorized keys '/home/jhd/.ssh/authorized_keys': No such
file or directory
```

有很多种方式可以用来分析日志记录，并从中寻找程序出错的原因。

- 可以使用系统自带的 GUI 事件查看器，并运用其搜索与过滤功能。
- 可以用编辑器打开并处理日志文件（参见第 24 条）。
- 可以用 Unix 工具来过滤、汇总并筛选相关的字段（参见第 22 条）。
- 可以对日志进行交互式的监控（参见第 23 条）。
- 可以使用 ELK、Logstash、loggly 或 Splunk 这样的日志管理应用程序或服务。
- 在 Windows 系统下，可以使用 Windows Events Command Line Utility（wevtutil）来执行查询并导出日志。

一般来说，我们总是先从与程序的出错时间较为接近的那些条目入手。此外，也可以在事件日志里面搜寻与错误有关的字符串，比如，如果某个命令执行失败了，那你就在日志里面搜这个命令的名字。无论采用哪种方法，等你找到了相关的条目之后，就应该继续检查该条目之前的那些内容，看看里面有没有错误信息、警告信息，或是不应该出现的条目。

对于表现不太明显的错误来说，通常应该把日志文件里面那些无关紧要的条目删掉，从而使包含重要信息的条目得以凸现。我们可以采用编辑器来删除日志的内容：Emacs 用户使用 delete-matching-lines regular-expression 形式的命令来删除，vim 用户使用 :g/regular-expression/d 形式的命令来删除，而 Eclipse 及 Visual Studio 的用户，则可以通过查找 / 替换功能来把那些整行文字都符合某个正则表达式并以 \n 结尾的内容删除。如果你使用的是 Unix 命令行，那么可以用管道来拼接 grep-v 命令，以便对日志文件进行过滤。

要点

- 通过查看日志文件来寻找应用程序出错的原因。
- 提升应用程序在记录信息时所用的详细程度，以便把程序失败的原因记录下来。
- 对日志文件进行配置及过滤，以缩减问题的排查范围。

第 57 条：对系统和进程所执行的操作进行性能评测

在调试与性能有关的问题时，我们首先应该做的，就是了解该系统的操作概况（而且通常也只能通过这种手段来调试此类问题）。通过系统的操作概况（profile），我们可以

看出它对资源的利用情况，进而找到其中不太正常或是有待优化的那一部分。要进行性能概要分析，第一步是从宏观层面上总览整个系统。有两种进程查看工具，可以显示出系统的 CPU 及内存使用情况，它们分别是 Unix 系统上面的 top 命令，以及 Windows 系统上面的 Task Manager（任务管理器）程序，如图 7.1 所示。在表现不正常的系统上面，如果 CPU 的使用率过高（例如，对于单核的 CPU 来说，90% 的使用率可以说是过高了），那就意味着我们必须对程序的处理逻辑进行分析，反之，若是 CPU 的使用率过低（例如，对于单核的 CPU 来说，10% 的使用率可以说是过低了），则说明程序有可能因为某些输入 / 输出（I/O）操作，而发生了一定的延迟。请注意，对于多核的 CPU 来说，系统所显示的使用率，通常是指所有 CPU 内核的平均值，因此，如果你要调试的是一个单线程的进程，那就应该把刚才说的那两个限值，与 CPU 的内核数量相除。比如，图 7.1 所展示的这两个系统，都用的是 8 核的 CPU，因此，在整个系统处于闲置的状态下，如果现在有某个进程对其中一个 CPU 内核的使用率是 100%，而对其余 7 个内核的使用率是 0%，那么平均下来，整个系统的 CPU 使用率就是 12%（也就是 100%/8）。

除了要关注 CPU 的使用率，还应该关注系统物理内存的使用情况。如果使用率过高（也就是说，接近于 100%），那么可能会导致系统无法继续分配内存，进而使程序出现错误，或是导致系统必须去执行虚拟内存分页操作，进而使系统的性能下降。在计算剩余内存的数量时，要注意：Linux 系统几乎总是会把可以使用的内存，全都用作缓冲区缓存，因此，对于 Linux 系统来说，我们在计算其剩余内存时，还应该把 top 程序在 buffers 字段中所给出的值也一起考虑。

有些系统在设计的时候，本身就是打算以使用率接近 100% 的方式来运作的，对于这种系统来说，我们不应该总是盯着使用率不放，而是还要考虑饱和度（saturation）的问题。也就是说，要看某项资源所处理的请求，是不是超过了它的服务能力。为此，我们还是可以使用相同的工具来监测，只不过这一次，要关注的是那些与具体资源有关的饱和度指标。

- 对于 CPU 来说，要看其负载是不是高于内核数量（适用于 Unix 系统）或 Performance Monitor-System-Processor Queue Length 中所显示的数值（适用于 Windows 系统）。
- 对于内存来说，要观察系统把虚拟内存页面写入磁盘的频率。
- 对于网络 I/O 来说，要观察丢包与重传的情况。
- 对于存储设备的 I/O 来说，要观察请求队列的长度及操作延迟。

非卖品，仅供非商业用途或交流学习使用

```
top - 21:38:21 up 51 days,  5:29,  1 user,  load average: 1.09, 1.00, 0.68
Tasks: 288 total,   2 running, 286 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.6 us,  0.1 sy,  0.0 ni, 87.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  12283536 total, 10967036 used,  1316500 free,  2328888 buffers
KiB Swap: 25071612 total,  1468356 used, 23603256 free.  6450376 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
31745 dds       20   0    4068    604    532 R 100.0  0.0  15:22.16 cpu-hog
 1623 rabbitmq  20   0 2225948  14584    632 S   0.3  0.1 672:13.14 beam.smp
 1665 postgres  20   0  110924     24      0 S   0.3  0.0   7:24.37 postgres
31557 root      20   0       0      0      0 S   0.3  0.0   0:00.06 kworker/6:1
31630 dds       20   0  114316   4052   3016 S   0.3  0.0   0:00.05 sshd
31746 dds       20   0   25740   2888   2272 R   0.3  0.0   0:01.27 top
    1 root      20   0   31160   3928   1952 S   0.0  0.0   1:01.43 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:04.90 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   1:28.10 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    7 root      20   0       0      0      0 S   0.0  0.0  19:02.24 rcu_sched
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S   0.0  0.0   0:02.42 migration/0
   10 root      rt   0       0      0      0 S   0.0  0.0   0:18.20 watchdog/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:17.61 watchdog/1
   12 root      rt   0       0      0      0 S   0.0  0.0   0:02.81 migration/1
   13 root      20   0       0      0      0 S   0.0  0.0   3:23.61 ksoftirqd/1
```

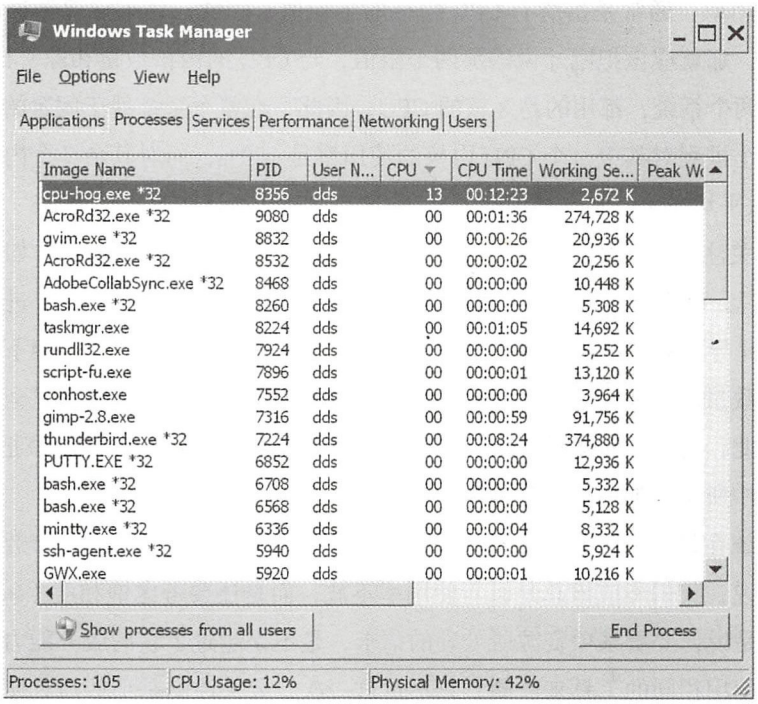


图 7.1 用 top 命令（顶部）及 Window 的 Task Manager 程序（底部）来查看当前正在运行的进程

就上述这些方面而言，如果饱和度看起来经常会超过 100%（无论是一直大于 100%，还是瞬间冲上 100%），那么通常就意味着系统出问题了。

对影响系统性能的各种因素有了宏观的了解之后，接下来，我们应该深入探查那些 CPU 周期占用数量过多、I/O 次数过多、I/O 延迟过大或内存用量过多的进程。

- 如果问题是 CPU 负载过高而引起的，那么就应该检查正在运行的这些进程，并按照它们对 CPU 的使用率进行排序，从而找出占用 CPU 时间最多的那个进程。

非卖品，仅供非商业用途或交流学习使用

在图 7.1 中，这个进程的名字叫做 cpu-hog。

- 如果问题是内存使用率过高而引起的，那就按照其工作集的内存大小（或常驻内存的大小）来对进程排序。排序时所依据的是其对物理内存的占用量，而不是对虚拟内存的占用量。
- 如果你怀疑问题是由于 I/O 负载过多或 I/O 延迟过大所引发的，那么可以使用 Unix 系统的 iostat、netstat、nfsstat 或 vmstat 等工具，以及 Windows 系统的 Performance Monitor 程序（在系统中运行 perfmon 命令，就可以打开该程序）来进行监测。要注意观察与磁盘及网络数据相对应的 I/O 操作数量，因为这两个方面都有可能成为系统性能的瓶颈。等你把引发性能问题的那种因素确定下来之后，就可以通过 Unix 系统的 pidstat 命令或 Windows 系统的 Task Manager 程序来找到与之对应的进程了。然后，查看该进程所执行的系统调用，以便深入地了解其行为（参见第 58 条）。

对于 CPU 或内存使用率过高的情况来说，我们还应该对有问题的进程做性能概要分析，以探查其行为。有很多技术都能够监测程序的行为。如果你关注的是 CPU 使用率，那就把程序放在 statistical profiler（统计式性能分析器）下面运行，这种 profiler，在每一秒的时间内，都会数次打断程序的操作，并且会把其中耗时最多的那些操作记录下来。此外，也可以对编译器或运行时系统进行设置，令其在每个（非内联的）函数开头及结尾处放置一些代码，并据此创建出一张与程序的执行概况有关的图表。这样一来，我们就可以把每个函数所做的事情，都计算到它的上层函数头上，从而厘清那些调用路径较为复杂的性能问题。如果采用 GCC 进行编译，那么需要开启 -pg 选项，并使用 gprof 工具来查看生成的数据。在极个别的场合，你甚至还可以令编译器通过计数器来对每一个基本的代码块进行统计，以了解每行代码分别执行了多少次。如果采用 GCC 进行编译，那么需要开启 -fprofile-arcs 和 -ftest-coverage 选项，并使用 gcov 工具来给代码添加相关的注解。除了这些办法，还可以考虑通过 Eclipse 及 NetBeans 的 profiler 插件来执行性能概要分析，或是采用针对 Java 程序的 VisualVM、JProfiler 和 Java Mission Control 系统（参见图 7.2），以及针对 .NET 代码的 CLR profiler 来进行分析。如果你关注的是内存使用率，那么通常可以用监控器来对运行时系统的内存分配操作进行修改，以便追踪内存的分配情况。Unix 系统下的 Valgrind 是个很有用的工具，刚才提到的 VisualVM 及 Java Mission Control，也属于此类工具。我们可以通过 AspectJ 和 Spring AOP 等面向方面的编程（Aspect Oriented Programming）工具及框架来自行安排并配置

非卖品，仅供非商业用途或交流学习使用

所要执行的监控。在更靠近硬件的层面上，我们还可以用 perf、oprofile 或 perfmon2 等工具来监测 CPU 的性能计数器，以了解缓存未命中、分支预测未命中，以及指令获取迟滞等情况（详情参见第 65 条）。

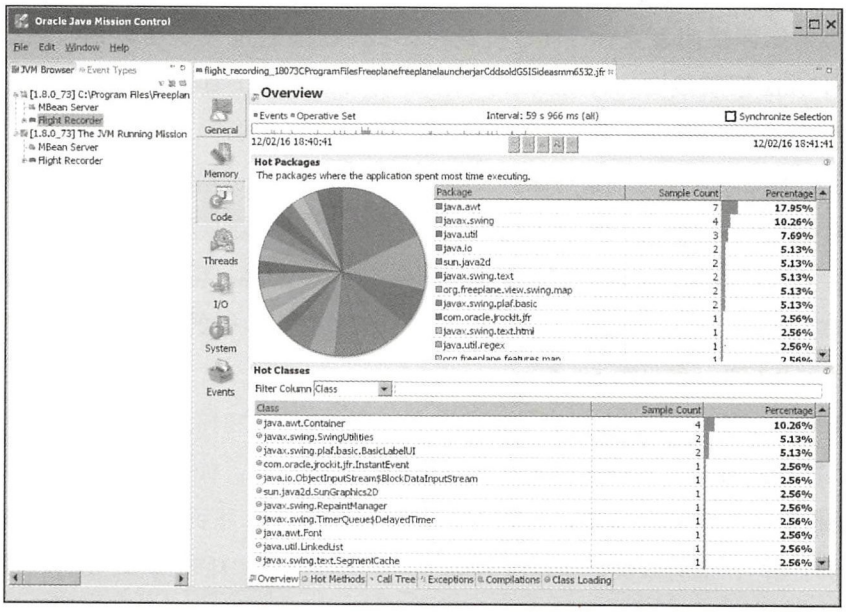


图 7.2　由 Java Mission Control 所提供的性能概况图，其中列出了耗时较多的包和类

要点

- 检查 CPU、I/O 及内存的使用率及饱和度，以便分析性能问题。
- 给有问题的进程做性能分析，以了解其对 CPU 及内存的使用情况，进而缩小需要排查的代码范围。

第 58 条：追踪程序的执行情况

有一些可以对程序进行监测及追踪的工具和设施，能够根据任何一个程序的执行情况来生成类似于日志的数据。与应用程序层面的日志记录相比，这种办法有下面几个好处：

❑ 即便正在调试的这款应用程序不提供日志机制，也依然能够获取到一些数据。

非卖品，仅供非商业用途或交流学习使用

- 无需专门去准备 debug 版本的软件（参见第 40 条），况且 debug 版本的软件有可能会掩盖最初的问题。
- 与带有图形界面的调试器相比，这种办法更加轻便，从而可以在没有安装相关工具的生产环境中使用。

我们通常是采用这样的办法来查找 bug 的：要么在程序的关键地点放置 log 语句（参见第 56 条和第 41 条），要么就把代码放在调试器里面运行，以便向其中动态地插入断点（参见第 30 条）。

然而当今有很多性能问题与程序 bug，都要涉及第三方的程序库，或是要涉及本程序与操作系统之间的交互。对于这些问题来说，有一种办法，是去查看本程序的代码究竟是如何调用另外一个组件的。我们可以检查每次调用的时间戳，或是看看有没有数量过多的调用行为，并以此来找寻性能问题。此外，函数的参数通常也可以揭示出某些 bug。适用于 Unix 系统的调用追踪工具包括 ltrace（用来追踪对程序库的调用）、strace、ktrace 以及 truss（这三个工具用来追踪对操作系统的调用）；适用于 Java 程序的工具是 JProfile；适用于 Windows 系统的工具是 Process Monitor（用来追踪对 DLL 的调用，这些调用既包括对操作系统接口的调用，也包括对第三方库接口的调用）。这些工具通常会使用特殊的 API 或通过代码补丁（code-patching）技术来把自己挂在待调试的程序与其外部接口之间。

现在举一个例子。假设现在有一个程序，在处理其输入数据的时候，执行得特别缓慢（参见第 10 条），而对该程序运行 strace 命令之后，得到了下面这样的输出，那么，我们就可以看出：I/O 程序库的缓冲区并没有得到利用，而且程序在处理每一个字符的时候，都会发出两次系统调用，一次是读取 8191 字节，另一次是把文件的搜寻指针后退 8190 字节。

```
read(6, "ng or publicity pertaining\nto di"..., 8191) = 8191
_llseek(6, -8190, [536], SEEK_CUR) = 0
read(6, "g or publicity pertaining\nto dis"..., 8191) = 8191
_llseek(6, -8190, [537], SEEK_CUR) = 0
read(6, " or publicity pertaining\nto dist"..., 8191) = 8191
_llseek(6, -8190, [538], SEEK_CUR) = 0
read(6, "or publicity pertaining\nto distr"..., 8191) = 8191
_llseek(6, -8190, [539], SEEK_CUR) = 0
read(6, "r publicity pertaining\nto distri"..., 8191) = 8191
_llseek(6, -8190, [540], SEEK_CUR) = 0
```

得知这一情况之后，我们可以去查看程序用来处理输入数据的那部分代码，而且还

非卖品，仅供非商业用途或交流学习使用

可以推想：这可能是由于程序调用了 tellg 方法。下面这个短小的测试程序（参见第 11 条），也能够表现出同样的问题：

```
ifstream in(fname.c_str(), ios::binary);

do {
  (void)in.tellg();
} while ((val = in.get()) != EOF);
```

如果我们能够通过上面这种方式来简单而可靠地重现问题，那么很容易就可以写出一个 shim 类，用来独立地计算当前位置距离文件开头的偏移量，并对该量进行缓存，从而无需再去调用 tellg 函数。

用 Unix 工具对 strace 所输出的内容进行处理，可以给调试工作带来很大的帮助。比如，如果有一个程序因为某个配置条目出错而无法运作，但我们又不方便在几十个配置文件里面手工寻找这个条目，那么就可以用下面这条 Bash 命令来把那些由 prog 程序所开启，且含有不当字符串（如 xyzzy）的文件列出来。

```
# Send the output of strace to a command
strace -fo >(
  # Isolate and output the path of each opened file
  sed -n 's/.*open("\(\/[^"]*\)".*= [^-].*/\1/p' |
  # Remove special device files
  egrep -v '^/(proc|dev|tmp)/' |
  # Output each file path only once
  sort -u |
  # Search for occurrences of xyzzy within each file
  xargs fgrep xyzzy) prog
```

上面这条命令把 strace 程序所输出的内容发送给一条管道，这条管道首先用 sed 命令把一些文件名提取出来，这些文件名曾经传给了名为 open 的系统调用，然后，它用 egrep-v 命令把与设备有关的文件名删掉，接下来，用 sort-u 命令对这些文件名进行排序，并去除重复的文件名，最后，用 fgrep 命令在剩下的文件里面寻找字符串 xyzzy。

如果要调试的是 Java 程序，或是使用了 X Window 系统的程序，那么在用 strace 查看其系统调用情况时，就会看到很多与运行时框架有关的信息，这些信息颇为杂乱，使人无法看清程序的真实行为。所幸我们可以用 strace 的 -e 选项来把这些系统调用过滤掉。下面这两条命令，分别适用于 Java 程序以及使用了 X Window 系统的程序：

```
# Trace a Java program
strace -e 'trace=!clock_gettime,gettimeofday,futex,\
timerfd_settime,epoll_wait,epoll_ctl'
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
# Trace a Unix X Window System program
strace -e 'trace=!poll,recvfrom,writev,read,write'
```

请注意，追踪工具也可以连接到正在运行的程序上面。命令行形式的追踪工具，提供了 -p 选项，而带有图形界面的追踪工具，则会提示你去单击想要追踪的进程。

我们并不是只能对与系统及程序库有关的调用进行追踪。绝大多数的解释型语言，也都提供了相应的选项，用来在程序执行期间追踪其行为。下面列出几种常见的脚本语言所提供的追踪选项：

- Perl:perl-d:Trace
- Python:python-m trace--trace
- Ruby:ruby-r tracer
- Unix shell:sh-x、bash-x、csh-x 等

除此之外，还有其他一些方式，也可以对程序所执行的操作进行监测，例如，可以使用针对 JavaScript 程序的后端监测工具 spy-js、使用与网络包有关的监测工具（参见第 16 条），或是通过数据库服务器来记录应用程序所执行的 SQL 语句等。比如，下面这几行 SQL 语句，就可以用来开启 MySQL 的日志记录功能：

```
set global log_output = 'FILE';
set global general_log_file='/tmp/mysql.log';
set global general_log = 1;
```

上面提到的那些工具，都是一些老牌的监测工具，只要你能够大致确定问题的原因，那么这些工具就可以在解决问题的过程中，给你提供很多的帮助。然而，它们同时也有着一些缺点，比如，它们总是要求开发者必须先对代码采取一些特别的措施，然后才能对其进行监控；它们会使系统的性能降低；它们的界面很古怪，而且互不兼容；它们只能展示出整个情境中的一小部分，而且有时会忽略重要的细节。

有一种工具能够避开这些缺陷，它的名字叫做 DTrace，这是个原本由 Sun 公司所开发的动态追踪框架，能够以一套统合的机制来全面而自然地监控操作系统、应用程序服务器、运行时环境、程序库，以及一般的应用程序。它目前可以运行在 Solaris、OS X、FreeBSD 及 NetBSD 系统上面。对于 Linux 系统来说，SystemTap 及 LTTng 也提供了类似的功能。

DTrace 曾获得《华尔街日报》的技术创新奖，毫无疑问，这不是那种只用几天工夫就拼凑出来的产品。Sun 公司的三位工程师花了好几年的时间来开发这套机制，使其能够安全地运用到所有的操作系统内核函数、动态链接库、应用程序的函数、特定的

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

CPU 指令以及 Java 虚拟机上面。他们还研发了一款安全的解释型语言，令开发者能够以此来编写复杂的追踪脚本，这种脚本不会损害操作系统的功能，而且能够在不耗费大量内存的前提下，灵活地将多个函数统合起来，以便对追踪到的数据进行汇总。DTrace 整合了现有的大多数追踪工具及某些知名的解释型语言，并具备了那些工具和语言所拥有的一些技术与奇妙的功能，进而成为了一套包罗万象的程序追踪平台。

我们通常是通过 dtrace 命令行工具来使用 DTrace 框架的。我们用一种叫做 D 语言的领域特定语言（它与同名的通用编程语言不是一回事）来撰写脚本，并将其传给 dtrace 工具，该工具会安装脚本中的追踪指令，然后执行程序，并打印出追踪的结果。D 语言是一种很简单的语言，它和 awk 及 sed 等工具，以及其他很多声明式的语言（declarative language）类似，也是由成对的模式与动作所构成的。模式（在 DTrace 的术语里面，叫做谓词）用来指定探测器，也就是用来指定你想要监测的事件。DTrace 预先定义了上千种探测器（早期的 Solaris 版本有 4 9979 种，在笔者所用的 OS X El Capitan 版本上面，有 17 7398 种），此外，像是应用程序服务器与运行时环境等系统程序，也可以定义它们各自的探测器，而且你还可以在程序或动态链接库中的任意位置上安插探测器。比如，下面这条命令：

```
dtrace -n 'syscall:::entry'
```

会在每一个系统调用的入口点放置探测器，并执行（默认的）动作，也就是说，只要有进程去执行系统调用，它就会把这次调用的名称及进程的 process-id 打印出来。我们可以用 Boolean 操作符，把谓词与其他变量结合起来，以指定更为复杂的追踪条件。

在刚才那条命令中，syscall 这个名字所指定的是一种提供器，也就是一种能够提供探测器的模块。可以想见，这个名叫 syscall 的提供器，所提供的应该是一些能够追踪系统调用的探测器，在笔者的操作系统上面，它能够对 500 种系统调用进行追踪。我们也可以用 syscall::open:entry 这样的名称来把具体的探测器安装在系统调用的入口处，以便对该调用进行追踪（在本例中，这个调用指的是 open）。DTrace 有数十种提供器，可以对性能统计机制、全部的内核函数、锁、系统调用、设备驱动、输入事件与输出事件、进程的创建及终止、网络栈的管理信息库（management information base，MIB）、调度器、虚拟内存操作、用户程序中的函数与任意的代码位置、同步原语、内核的统计机制以及 Java 虚拟机的操作进行追踪。下面这些命令能够列出当前可以使用的提供器与探测器。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
# List all available probes
dtrace -l
# List system call probes
dtrace -l -P syscall
# List the arguments to the read system call probe
dtrace -lv -f syscall::read
```

我们可以在使用每个谓词的时候，为其定义一项动作。如果谓词的条件得以满足，那么 DTrace 就会执行这项动作。例如，下面这条命令：

```
dtrace -n 'syscall::open:entry {trace(copyinstr(arg0));}'
```

就会列出每一个已经打开的文件所具有的文件名。

动作也可以定义得复杂一些，比如，我们可以在其中设置全局变量或线程的局部变量，可以把数据存放在关联数组中，也可以用计数、求最小值、求最大值、求平均值以及量化等函数来对数据进行聚合。比如，下面这个程序，会把每条线程在 dtrace 执行期间得到调用的次数统计出来：

```
proc:::exec-success { @proc[execname] = count()}
```

有些函数可以获取资源，有些函数可以释放资源，如果我们将这两种函数配套使用，那么就可以轻松地调试资源泄漏问题。常见的 DTrace 脚本是可长可短的，短的脚本就像刚才那样，只有一行，而长的脚本，则可以有数十行代码，其中能够包含很多相互配对的谓词与动作。

如果要对运行在 JVM 上面的代码进行调试，那么还有另外一种工具也能够很好地追踪其行为，这就是 Byteman。该工具能够在无需重新编译代码的前提下，向应用程序的方法或 Java 的运行时系统中注入 Java 代码。你可以通过清晰而简单的脚本语言来告诉该工具何时应该对原始的 Java 代码进行转换，并对转换的方式做出规定。与在代码里面手工添加 log 语句相比，使用 Byteman 有三个好处。首先，由于我们无需获得受测程序的源代码，因此不仅可以用它来调试自己所写的程序，而且也可以用它来调试第三方的程序。其次，我们可以故意注入一些错误的内容或类似的状况，看看程序会做何反应。最后，我们还可以在 Byteman 脚本中，检查受测程序当前的内部状态与它本来应该处在的状态是否相符，并且在二者不相符时，令该测试用例失败。

对于 Windows 平台来说，Windows Performance Toolkit 也提供了类似的功能，它是 Windows Assessment and Deployment Kit 的一部分。这个 Toolkit 中有一个名为 Windows

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Performance Recorder 的记录组件，可以在性能有问题的系统上面，对你认为重要的那些事件进行追踪，此外，还有一个名为 Windows Performance Analyzer 的组件，它完全采用 Windows 系统的风格，通过漂亮的图形界面来在图表中展示分析结果，并允许我们对这些表格进行操作。

要点

- 我们可以在不访问源代码的前提下来追踪系统与程序库的调用情况，并监测程序的行为。
- 要学会使用 Windows 系统的 Windows Performance Toolkit、Linux 系统的 SystemTap，以及 OS X、Solaris、FreeBSD 系统的 Dtrace 等工具。

第 59 条：使用动态程序分析工具

有很多专门的工具可以对编译后的程序进行修改，向其中注入一些检测例程，而且可以对程序的执行情况进行监控，并把有可能出错的地方汇报给用户。由于这种检测是在程序运行的时候执行的，因此称为动态分析。这些动态分析技术，与我们在第 51 条中所说的那些静态分析技术，是可以相互补充的，而且第 51 条里面提到的某些做法，例如，在编写 JavaScript 程序时使用的 "use strict"; ，以及在编写 Perl 程序时使用的 use strict; 和 use warnings; 等，其实相当于同时开启了静态检查与动态检查功能。此外，第 53 条里面提到了一些技术，可以把受测程序与调试用的程序库相链接。这里所说的动态分析技术，可以与那些链接技术互为补充，而且那些技术也会同时开启某些动态检测功能。

与静态分析工具相比，动态分析工具更容易检测到实际上有可能发生的错误，因为它们是直接在程序执行代码时进行追踪，而不是像静态工具那样，去推测程序有可能会执行什么样的代码。这意味着动态分析工具所发现的错误，几乎不太可能是误报。然而，由于动态分析工具只会关注程序实际执行的代码，因此，它或许会忽略尚未执行到的那些代码路径里面所包含的错误，于是，就有可能发生大量的漏报。

由于动态分析工具通常会极大地减缓程序的执行速度，并且会报告出很多不太重要的错误，因此，最好是能够先编写一段特别具体的测试脚本，把当前正在调试的那个问题精确地重现出来，然后再去使用此类工具。或者也可以把有待分析的代码，放在一个真实且完备的测试场景中运行，以此来保持代码的整洁，因为在这个过程中，我们可以

非卖品，仅供非商业用途或交流学习使用

把动态分析工具所汇报的全部错误，都添加到白名单里面，从而能够在修改代码之后，立刻发现新引入的那些错误。

许多动态分析工具都提供了一些检测机制，用来检查程序有没有使用未经初始化的变量、有没有泄漏内存，以及有没有访问可用内存空间以外的地址。还有其他一些动态分析工具，可以用来寻找安全漏洞、尚未完全优化的代码、处在受测范围之外的代码（这表示程序的测试工作有疏漏）、隐式的类型转换操作、动态类型不一致，以及数值溢出等。此外，还可以参考第 62 条中所说的技术，采用动态分析工具来捕获并发方面的错误。维基百科的 dynamic program analysis 词条列出了几十种工具，你可以根据自己所在的环境、所面对的问题以及所拥有的预算来从中选择合适的工具。

有一种流行的开源工具包，叫做 Valgrind，它是个动态分析系统，其中含有一个功能强大的内存检测组件。现在考虑下面这个程序，该程序只用 3 行代码就展示出了 3 种错误，它们分别是：泄漏内存、访问非法的内存地址，以及返回未经初始化的变量。

```
#include <stdlib.h>

int
main()
{
      char *c = malloc(42);

      c[42] = 1;
      return c[0];
}
```

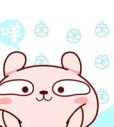
如果我们用下面这条命令来运行上述程序，

```
valgrind --track-origins=yes --leak-check=yes memory
```

那么就会看到下面这样的输出结果，其中列出了全部的三项错误，以及与每项错误相对应的程序位置：

```
Invalid write of size 1
  at 0x400524: main (memory.c:8)
 Address 0x51de06a is 0 bytes after a block of size 42 alloc'd
  at 0x4C28C20: malloc (vg_replace_malloc.c:296)
  by 0x400517: main (memory.c:6)

Syscall param exit_group(status) contains uninitialised byte(s)
  at 0x4EECAB9: _Exit (_exit.c:32)
  by 0x4E6CB8A: __run_exit_handlers (exit.c:97)
  by 0x4E6CC14: exit (exit.c:104)
```



非卖品，仅供非商业用途或交流学习使用


```
  by 0x4E56B4B: (below main) (libc-start.c:321)
 Uninitialised value was created by a heap allocation
  at 0x4C28C20: malloc (vg_replace_malloc.c:296)
  by 0x400517: main (memory.c:6)
HEAP SUMMARY:
   in use at exit: 42 bytes in 1 blocks
 total heap usage: 1 allocs, 0 frees, 42 bytes allocated

42 bytes in 1 blocks are definitely lost in loss record 1 of 1
  at 0x4C28C20: malloc (vg_replace_malloc.c:296)
  by 0x400517: main (memory.c:6)

LEAK SUMMARY:
  definitely lost: 42 bytes in 1 blocks
  indirectly lost: 0 bytes in 0 blocks
    possibly lost: 0 bytes in 0 blocks
  still reachable: 0 bytes in 0 blocks
      suppressed: 0 bytes in 0 blocks

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

还有一个值得注意的动态分析框架，叫做 Jalangi，它适用于客户端及服务器端的 JavaScript 程序。该框架可以对 JavaScript 代码进行转化，令其通过 API 来展示自己的执行情况，从而使我们能够编写脚本来对其在发生特定事件时的执行情况加以验证，比如，可以验证二元算术操作的求值情况。这种脚本可以探查 JavaScript 代码中的各种问题。现在考虑下面这段代码，它会产生一个无效的数，在 JavaScript 中，这也称为非数（not a number）或 NaN。

```
var a = "a";
var b = a * 2;
console.log(b);
```

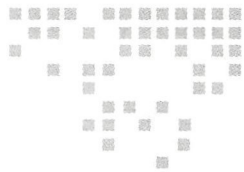
如果我们用该工具的作者所提供的检测器来撰写脚本，并将其放在 Jalangi 之下运行，那么就会看到下面这样的错误：

```
[Location: line No.: 2, col: 9]
   binary operation leads to NaN:NaN <- a [string] * 2 [number]
[Location: line No.: 2, col: 9] writing NaN value to
   variable:b: NaN
[Location: line No.: 3, col: 13] read NaN from variable b :NaN
```

要点

- 用动态程序分析工具来寻找代码运行过程中的实际问题。

非卖品，仅供非商业用途或交流学习使用



第 8 章 Chapter 8

调试多线程的代码

CPU 制造商会在芯片所能容纳的前提下把越来越多的晶体管放入多个核心中，并请求开发者善用这些核心。如果运行在多个 CPU 内核上的各条执行线程彼此不进行协调，那么它们就会各自按照最快的方式执行。而这样做，会令程序的执行效果显得不那么确定，因为每次运行程序的时候，这些线程的执行顺序都会和上一次稍有区别。于是，本书其他各章所讲解的许多技术，在多线程环境下都会失效。比如，由于我们要寻找的 bug 总是飘忽不定，因此很难像第 4 条所说的那样，采用可靠的办法来详细观察该 bug。所幸有各种工具和技巧，可以专门用来解决此类问题。

请注意，本章所提到的许多技巧，针对的都是那些明确使用了低级并发组件的代码，例如，操作系统、数据库、游戏引擎或程序库等系统级的代码，以及那些有待维护的遗留软件中的代码。对于应用程序级别的软件来说，如果要从头开始制作，那么大多数情况下，都应该直接使用高级的并发抽象机制，而不应该使用低级的并发组件（参见第 66 条）。如果你感觉自己在调试这些新写的并发代码时花的时间太多了，那么可能要想一想你的编程方式是不是有错。

第 60 条：通过事后调试来分析死锁问题

如果有一组线程，其中的每条线程都在获取了自己所需的资源之后，想要继续获取

非卖品，仅供非商业用途或交流学习使用

由其他线程所占据的资源，那么这组线程就会陷入无尽的等待中，从而无法继续运作，这就是死锁现象。比如，现在有两个共用的数据结构，分别为甲、乙两条线程所占据，而这两条线程都必须继续锁定另外一个数据结构才能够向下执行，于是，它们就会不停地试着去获取由对方所占据的那个数据结构，从而陷入僵局。此时，我们就说，这两条线程发生了死锁。据说，Kansas 的议会有过这样一个提议，它极好地说明了什么叫做死锁：

“两辆火车在铁道口相遇时，都应该完全停下，直到另外一方走后，才可以再度出发。”

《ACM Queue》上有一篇文章《Real-world Concurrency》，其作者 Bryan Cantrill 与 Jeff Bonwick 认为：有很多谈论并发问题的人都把死锁描绘得过于可怕了，其实在并发方面的各种 bug 中，它是最容易分析的一类。因为死锁的系统，实际上就是一个处在冻结状态的系统，于是，我们可以将其状态保存到核心转储文件里面（参见第 35 条），然后对该文件进行分析，以寻找当时陷入等待状态的那些线程以及它们所执行到的位置。通过该信息，我们可以发现与死锁有关的实体（如两个或多个锁），并了解其所对应的错误，这种错误一般是指多个线程在某一组资源上陷入了循环等待。

程序清单 8.1　会陷入死锁的 C++ 程序

```
#include <iostream>
#include <mutex>
#include <thread>
#include <unistd.h>

using namespace std;

static mutex m1, m2;

void bob()
{
        lock_guard<mutex> g1(m1);
        sleep(1);
        lock_guard<mutex> g2(m2);
        cout << "Hi, it's Bob" << endl;
}

void alice()
{
        lock_guard<mutex> g1(m2);
        sleep(1);
        lock_guard<mutex> g2(m1);
        cout << "Hi, it's Alice" << endl;
}
```

非卖品，仅供非商业用途或交流学习使用


```
25
26  int main()
27  {
28        thread t_bob(bob);
29        thread t_alice(alice);
30
31        t_bob.join();
32        t_alice.join();
33        return 0;
34  }
```

现在我们来看程序清单 8.1 中的这个 C++ 程序（该程序遵从 C++11 标准）。bob 函数和 alice 函数分别运行在两条线程中，它们都必须先获取 m1 和 m2 这两把互斥锁，然后才能够继续执行。于是，这个 C++ 程序偶尔就有可能发生冻结。如果你想更明确地观察这种效果，那么可以把该程序放到下面这个 shell 循环里面运行：

```
while true ; do
  deadlock
done &
```

当你发现某个程序出现冻结状况之后，可以按照下面这套步骤去调试它。这套步骤是用 Unix 平台中的工具和方法演示的，然而也可以运用到其他平台上面。

假设这个程序是在客户所在的地方发生冻结的，那么第一步应该获取该程序的核心转储文件。我们需要用 ps 命令找到程序所对应的 process-id，然后用 kill 命令给具有这个 id 的程序发送 QUIT 信号。

```
$ ps
  PID TTY      TIME CMD
23010 pts/0 00:00:00 bash
23153 pts/0 00:00:00 bash
23175 pts/0 00:00:00 deadlock
23203 pts/0 00:00:00 ps
$ kill -QUIT 23175
-bash: line 15: 23175 Quit (core dumped) deadlock
```

然后，我们把从客户那里收集到的核心转储文件复制过来，并在它上面运行 gdb 命令。由于这份文件要反映出进程的内存布局，因此，可能会占据好几 MB 的空间（本例中的这份文件在 Linux 系统上是 17MB，在 FreeBSD 系统上是 9MB），不过其中大部分内容可能都是空白，于是，很容易就能将其压缩到几 KB，并通过电子邮件发送过来。在调用 gdb 命令时，要指定可执行程序的镜像名称，以及内存转储文件的名称。

```
gdb deadlock core
```

然后，列出当时正在执行的线程。

```
(gdb) info threads
  Id   Target Id         Frame
  3    Thread 0x7f6be668c700 (LWP 22096) __lll_lock_wait ()
  2    Thread 0x7f6be6e8d700 (LWP 22095) __lll_lock_wait ()
* 1    Thread 0x7f6be7e7e740 (LWP 22094) 0x00007f6be72404db
   in pthread_join (threadid=140101412247296, thread_return=0x0)
```

现在我们可以看到，这两个线程似乎都在等着获取某一把锁。如果程序更为复杂，那么可能只会看到其中的某一部分线程在等待某种资源，而不是像本程序这样，所有的线程都在等待着资源。对于那种程序来说，只需要关注正在等待资源的那些线程就可以了。

接下来，我们根据刚才在最左边那一栏中显示出的线程 ID 来从中选择一个正在等待资源的线程，并将其 ID 传给 thread 命令。

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x7f6be6e8d700 (LWP 22095))]
#0 __lll_lock_wait ()
   at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
```

把该线程的栈帧打印出来，以便寻找程序究竟是在运行到哪一行代码的时候试图去获取这把锁的。

```
(gdb) backtrace
#0 __lll_lock_wait ()
   at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1 0x00007f6be72414b9 in _L_lock_909 ()
  from /lib/x86_64-linux-gnu/libpthread.so.0
#2 0x00007f6be72412e0 in __GI___pthread_mutex_lock (mutex=
   0x6046c0 <m2>) at ../nptl/pthread_mutex_lock.c:79
#3 0x0000000000040109e in __gthread_mutex_lock (__mutex=0x6046c0
   <m2>) at /usr/include/x86_64-linux-gnu/c++/4.9/bits/
   gthr-default.h:748
#4 0x000000000040141a in std::mutex::lock (this=0x6046c0 <m2>)
   at /usr/include/c++/4.9/mutex:135
#5 0x00000000004015ce in std::lock_guard<std::mutex>::lock_guard
   (this=0x7f6be6e8ce40, __m=...) at /usr/include/c++/4.9/mutex:
   377
#6 0x0000000000401192 in bob () at deadlock.cpp:12
#7 0x0000000000402785 in std::_Bind_simple<void (*())()>::
   _M_invoke<>(std::_Index_tuple<>) (this=0x2438038) at
   /usr/include/c++/4.9/functional:1700
   [...]
#12 0x00007f6be6f7404d in clone ()
```

仔细查看刚才的程序清单，我们就会发现，问题出在 deadlock.cpp 的第 14 行代码上。

接下来，针对陷入死锁状态的其他线程，执行同样的操作。

```
(gdb) thread 3
[Switching to thread 3 (Thread 0x7f6be668c700 (LWP 22096))]

#0 __lll_lock_wait ()
   at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
(gdb) backtrace
#0 __lll_lock_wait ()
   at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1 0x00007f6be72414b9 in _L_lock_909 ()
  from /lib/x86_64-linux-gnu/libpthread.so.0
#2 0x00007f6be72412e0 in __GI___pthread_mutex_lock (mutex=
  0x604680<m1>) at ../nptl/pthread_mutex_lock.c:79
#3 0x000000000040109e in __gthread_mutex_lock (__mutex=0x604680
  <m1>) at /usr/include/x86_64-linux-gnu/c++/4.9/bits/
  gthr-default.h:748
#4 0x000000000040141a in std::mutex::lock (this=0x604680 <m1>)
   at /usr/include/c++/4.9/mutex:135
#5 0x00000000004015ce in std::lock_guard<std::mutex>::lock_guard
   (this=0x7f6be668be40, __m=...) at /usr/include/c++/4.9/mutex:
   377
#6 0x000000000040122f in alice () at deadlock.cpp:19
#7 0x0000000000402785 in std::_Bind_simple<void (*())()>::
   _M_invoke<>(std::_Index_tuple<>) (this=0x2438038) at
   /usr/include/c++/4.9/functional:1700
   [...]
#12 0x00007f6be6f7404d in clone ()
```

上述程序清单中的信息告诉我们：另外一条线程卡在了 deadlock.cpp 文件的第 22 行。查看该行附近的代码之后，我们就会发现，之所以会产生死锁，是因为 bob 函数在获取了 m1 之后，还想继续获取 m2（这发生在 deadlock.cpp 文件的第 14 行），而另一方面，alice 函数则在获取了 m2 之后，还想继续获取 m1（这发生在 deadlock.cpp 文件的第 22 行）。

如果系统有可能发生死锁，那么其设计者就应该制订一套处理办法。可以用来应对死锁问题的方式包括忽视死锁、检测死锁、预防死锁或避免死锁等。在调试死锁问题的时候，应该先了解正在调试的这个系统到底是采用哪种方式来应对死锁的。为此，我们可以查阅该系统的文档。然而更为务实的办法，则是去检查它的代码，看看其中会不会出现可能引发死锁的状况。如果你碰巧是第一个发现该系统会出现死锁的人，那么你应该做的事情，通常是自己来设计一套处理死锁问题的方案。你可以尽量缩减锁的数

量、避免重入、建立锁定体系，或是在代码中使用适当的同步原语，令程序在有可能发生死锁时，无法获取到相应的锁，从而避免其陷入阻塞。如果这些方案都行不通，那么你就必须根据系统对死锁问题的处理方式来修改自己的代码了。对于本例来说，较为实用的一种做法，是对获取资源的顺序进行调整，以确保这些资源总是按照某种确定的顺序来进行锁定（调整之前的那种顺序，有一个专门的称呼，叫做偏序，可以用来描述较为复杂的资源依赖关系）。于是，我们只需要把 alice 函数的加锁顺序调换一下就行了。

```
lock_guard<mutex> g1(m1);
lock_guard<mutex> g2(m2);
cout << "Hi, it's Alice" << endl;
```

程序清单 8.2　可能陷入死锁的 Java 程序

```
 1 public class Deadlock {
 2    public static void main(String[] args) {
 3       Object mutex1 = new Object();
 4       Object mutex2 = new Object();
 5
 6       Runnable bob = () -> {
 7          for (int i = 0; i < 1000; i++)
 8             synchronized(mutex1) {
 9                synchronized(mutex2) {
10                   System.out.println("Hi, it's Bob " + i);
11                }
12             }
13       };
14       Runnable alice = () -> {
15          for (int i = 0; i < 1000; i++)
16             synchronized(mutex2) {
17                synchronized(mutex1) {
18                   System.out.println("Hi, it's Alice " + i);
19                }
20             }
21       };
22
23       Thread at = new Thread(alice);
24       Thread bt = new Thread(bob);
25
26       bt.start();
27       at.start();
28
29       try {
30          at.join();
31          bt.join();
32       } catch(InterruptedException e) {
```

```
33            System.err.println("Interrupted: " + e);
34        }
35    }
36 }
```

现在我们再举一个例子，看看怎样调试 Java 程序的死锁问题。程序清单 8.2 所发生的死锁现象，与前面那个例子类似，它通常也会在打印了几次“ Hi it's Bob”之后，就陷入阻塞。所幸 Oracle 的 Java Development Kit 里面提供了 jstack 命令，可以较为容易地找到死锁的原因。我们需要查询陷入死锁的那个进程所对应的 process-id，然后把这个 ID 作为参数，传给 jstack，以显示出与死锁有关的信息及相关的代码。

```
$ jps
5504 Deadlock
8848 Jps
$ jstack -l 5504
2016-02-15 18:37:53
Full thread dump Java HotSpot(TM) Client VM
Found one Java-level deadlock:
=============================
"Thread-0":
  waiting to lock monitor 0x00d66d64 (object 0x0487f930,
  a java.lang.Object), which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00d66dd4 (object 0x0487f938,
  a java.lang.Object), which is held by "Thread-0"

Java stack information for the threads listed above:
===================================================
"Thread-0":
        at Deadlock.lambda$main$1(Deadlock.java:18)
        - waiting to lock <0x0487f930> (a java.lang.Object)
        - locked <0x0487f938> (a java.lang.Object)
        at Deadlock$$Lambda$2/29293983.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)
"Thread-1":
        at Deadlock.lambda$main$0(Deadlock.java:10)
        - waiting to lock <0x0487f938> (a java.lang.Object)
        - locked <0x0487f930> (a java.lang.Object)
        at Deadlock$$Lambda$1/19011157.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)

Found 1 deadlock.
```

要点

- 要想调试死锁问题，就应该对死锁的程序做一份快照，看看究竟有哪些线程和

代码，在争夺同一套资源时陷入了僵局。

第 61 条：捕获并重现

有一种效果很好的办法，是把程序在运行过程中所执行的操作详细记录下来，并据此来应对那些含有不确定因素的 bug。Intel 的 PinPlay/DrDebug Program Record/Replay Toolkit 就属于此类工具，它针对的是 Intel 架构的二进制文件。该工具可以与 Eclipse 或 gdb 结合起来使用。还有一种能够应对 Java 程序的记录器与调试器，叫做 Chronon。这些工具会详细地记录程序在运行过程中所发生的事件，使你能够看到这些操作究竟是以什么样的顺序来在多个内核上面并发执行的。例如，它们会把程序对主内存的读取和写入行为记录下来，以反映程序在执行相关操作时所依照的实际顺序。

我们可以按照下列步骤来捕获并重现系统中的 bug。

1. 开启记录功能，并反复运行应用程序，直到能够重现 bug。如果 bug 的发生概率比较低，那么可以把程序放在 shell 脚本里面反复运行（参见第 60 条），令 bug 能够尽快表现出来。等 bug 表现出来之后，我们把记录到的结果保存起来，以便进行后续的分析。

2. 对记录到的结果进行分析，找到 bug 最先出现的地点。在该步骤中，可以使用一种叫做程序切片分析（program slice analysis）的技术来协助我们找出线程之间比较可疑的那些依赖关系。看看自己所使用的工具是否支持这项技术。

3. 把程序放在调试器中运行，并对把原来所记录到的结果进行重放，直至程序运行到出现 bug 的那个地方。

4. 对程序在该点的状态进行分析，以寻找 bug 背后的错误原因。

程序清单 8.3　带有竞争条件的程序

```
#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define C(x) assert((x) == 0)

static int counter;

void *increment(void *threadid)
{
```

非卖品，仅供非商业用途或交流学习使用

```
        int i, tmp;

        for (i = 0; i < 100000; i++) {
              tmp = counter;
              tmp++;
              counter = tmp;
        }
        return (NULL);
}

int main()
{
        pthread_t tid[2];
        int i;

        for (i = 0; i < 2; i++)
              C(pthread_create(&tid[i], NULL, increment, NULL));
        for (i = 0; i < 2; i++)
              C(pthread_join(tid[i], NULL));
        printf("counter=%d\n", counter);
        return 0;
}
```

现在来看一个具体的例子。程序清单 8.3 中的代码会出现竞争条件的问题，因为它在执行与 counter 变量有关的读取、递增及写入操作时，并没有将这些操作合起来视为一项原子操作。多运行几次程序，就可以看出这个问题。

```
$ ./race
counter=100000
$ ./race
counter=103754
$ ./race
counter=101233
$ ./race
counter=100000
$ ./race
counter=103977
```

为了执行上面提到第一个步骤，我们需要用 PinPlay 把有问题的运行过程记录下来。此外，你也可以用相应的 Eclipse 插件来执行类似的操作。如果要用 gdb_record 进行记录，那么就在程序进入 main 函数时，执行 pin record on 命令。要是程序在这次运行的过程中没有表现出故障，那就重复此过程，直到故障表现出来。程序在接受记录时，其运行速度至少要比 debug 版本慢两个数量级，因此，如果程序较为复杂，那么应该尽量晚一些开始记录。

非卖品，仅供非商业用途或交流学习使用

```
$ gdb_record race
(gdb) break main
Breakpoint 1 at 0x400730: file race.c, line 28.
(gdb) continue
Continuing.
Breakpoint 1, main () at race.c:28
28          for (i = 0; i < 2; i++)
(gdb) pin record on
monitor record on
Started recording region number 0
(gdb) continue
Continuing.
counter=127873
[Inferior 1 (Remote target) exited normally]
(gdb) quit
```

记录下来的数据会存放在 pinball 目录中，我们可以用 replay 命令来重放。由于程序在重放的过程中，会按照和当初相同的顺序来操作内存，因此，也会表现出同样的错误行为。

```
$ replay pinball/log_0
counter=127873
$ replay pinball/log_0
counter=127873
$ replay pinball/log_0
counter=127873
```

然后，我们需要根据记录下来的数据，把程序放在 gdb 调试器中运行，并监控第 16 行代码里面的 tmp 变量。

```
$ gdb_replay pinball/log_0 ./race
0x0000000000400737 in main () at race.c:28
28          for (i = 0; i < 2; i++)
(gdb) pin trace tmp at 16
monitor trace [ 0x4006ed : %rsp + -12 ] 4 at 0x4006fe #tmp:<16>
Tracepoint #1: trace memory [0x4006ed : %rsp offset -12 ]
length 4 at 0x4006fe #tmp:<16>
(gdb) break _exit
Breakpoint 1 at 0x7f09dbab22d0: _exit. (2 locations)
(gdb) continue
Continuing.
counter=127873

Breakpoint 1, __GI__exit (status=status@entry=0)
   at ../sysdeps/unix/sysv/linux/_exit.c:28
(gdb) pin trace print to tmp.txt
monitor trace print to tmp.txt
(gdb) quit
```

非卖品，仅供非商业用途或交流学习使用

程序执行完之后，生成的追踪信息会存放在 tmp.txt 文件里面。该文件的内容如下所示。大家可以想见：如果程序编写得没有问题，那么变量的值（也就是等号右边的那个值）总是应该持续增加。

```
0x00000000004006fe: [ 0x0004006ed:rsp + -12] = 0x7f09 #tmp:<16>
0x00000000004006fe: [ 0x0004006ed:rsp + -12] = 0x1 #tmp:<16>
0x00000000004006fe: [ 0x0004006ed:rsp + -12] = 0x2 #tmp:<16>
0x00000000004006fe: [ 0x0004006ed:rsp + -12] = 0x3 #tmp:<16>
0x00000000004006fe: [ 0x0004006ed:rsp + -12] = 0x4 #tmp:<16>
0x00000000004006fe: [ 0x0004006ed:rsp + -12] = 0x5 #tmp:<16>
```

拿到这份追踪信息之后，我们就可以在其中寻找重复的 tmp 值，如果能找到这样的值，那就说明程序在这里出了问题。下面这条 shell 命令，会提取每行文本的第 7 个字段（也就是代表 tmp 值的那个字段），对其排序，寻找重复的值，并把开头的几个值打印出来。

```
$ cut -d' ' -f7 tmp.txt | sort | uniq -d | head
0x108e5
0x108e6
0x108e7
```

从追踪信息里面找到重复出现的 tmp 值（如 0x108e5）之后，就可以执行 pin break 16 if tmp==0x108e5 命令来添加条件断点了。这种断点，会使得程序的重放过程在 tmp 值出现重复时停下来。

```
$ gdb_replay pinball/log_0 ./race
0x0000000000400737 in main () at race.c:28
28          for (i = 0; i < 2; i++)
(gdb) pin break 16 if tmp == 0x108e5
monitor break at 0x4006fe if [ 0x4006ed : %rsp + -12 ]
4 == 0x108e5 #tmp:<16>
Breakpoint #1: break at 0x4006fe if [0x4006ed :$rsp offset -12]
length 4 == 0x108e5 #tmp:<16>
(gdb) continue
Continuing.
Triggered breakpoint #1: break at 0x4006fe if [0x4006ed :$rsp
offset -12 ] length 4 == 0x108e5 #tmp:<16>[New Thread 13259]

Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 13259]
increment (threadid=0x0) at race.c:16
16                tmp = counter;
```

重放的过程中，如果程序停在了你早前所指定的断点处，那你就应该查看每一条线程及其所拥有的变量值，进而了解问题发生的原因。从下面这段调试会话中可以看到，

非卖品，仅供非商业用途或交流学习使用

两个线程所拥有的 tmp 变量，其取值是相同的，这说明设定 tmp 变量的操作，与更新 counter 变量的操作，本来应该视为一项原子操作，完整地执行，而不应该容许其中发生线程切换现象。

```
(gdb) print tmp
$1 = 67813
(gdb) info threads
[New Thread 13258]
  Id   Target Id         Frame
  3    Thread 13258      0x000000000040070e in increment
    (threadid=0x0) at race.c:18
* 2    Thread 13259      increment (threadid=0x0) at race.c:16
  1    Thread 13248      0x00007f09dbdbf66b in pthread_join (
    threadid=139680249001728, thread_return=0x0)
    at pthread_join.c:92
(gdb) thread 3
[Switching to thread 3 (Thread 13258)]
#0  0x000000000040070e in increment (threadid=0x0) at race.c:18
18                      counter = tmp;
(gdb) print tmp
$2 = 67813
```

要点

- 要想探查那种带有不确定因素的并发错误，首先应该把有问题的执行过程记录下来，然后对记录到的信息进行分析，接下来根据捕获到的记录文件，在调试器里面对程序的执行过程进行重放。

第 62 条：用专门的工具来探查死锁与竞争条件问题

编写底层的多线程代码，是一项很容易出错的工作，它的风险，几乎和骑着大象在冰面上走动一样。在编写这种代码的过程中，你会引入很多微妙的错误，而且每一个这样的错误，都有可能在即将发布演示版，或是即将到达交工期限的时候，突然发生在客户的计算机上面。如果我们能够通过工具来找到这些并发错误，那就能省下很多应付此类问题的时间。

有一种办法是对代码执行静态分析，从中寻找可能引发 bug 的模式（参见第 51 条）。比如，FindBugs 工具就可以找到 45 种与“多线程的正确性”有关的错误。这些错误可以揭示出不正确的同步元素、未匹配的 wait() 与 notify()、不一致的同步、未加

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

保护的字段以及未能释放的锁等问题。FindBugs 工具既可以从命令行、图形界面及 IDE 里面运行，也可以纳入持续集成的过程中。

现在考虑下面这段代码：

```
1  class Counter {
2     private int n = 0;
3
4     public synchronized void increment() {
5        n++;
6     }
7
8     public void decrement() {
9        n--;
10    }
11
12    public synchronized int value() {
13       return n;
14    }
15 }
```

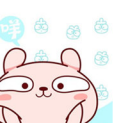
以及程序清单 8.4 中的这个程序。该程序会分别用两个线程来对计数器执行 10 0000 次递增和递减操作。

程序清单 8.4　在多个线程中使用同一个计数器

```
class ExerciseCounter {
   public static void main(String[] args) {
      final Counter c = new Counter();
      final int ITERATIONS = 100000;
      Thread inc = new Thread() {
         public void run() {
            for (int i = 0; i < ITERATIONS; i++)
               c.increment();
         }
      };

      Thread dec = new Thread() {
         public void run() {
            for (int i = 0; i < ITERATIONS; i++)
               c.decrement();
         }
      };

      inc.start();
      dec.start();
      try {
         inc.join();
         dec.join();
```



非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            } catch (InterruptedException e) {
                System.err.println(e);
            }
            System.out.println(c.value());
        }
    }
```

上述程序运行之后，通常会打印出负值（如 –6775）而不是 0。（笔者觉得大家应该能看出其中的 bug，然而你能不能解释一下，它打印出来的为什么经常是负数而不是正数呢？）

如果你编译 Counter 类，并在编译出来的 class 文件上面运行 FindBugs 工具，那么就可以看出：decrement 方法缺少 synchronized 关键字。

```
$ java -jar findbugs.jar -textui Counter.class
M M IS: Inconsistent synchronization of Counter.n;
locked 60% of time
Unsynchronized access at Counter.java:[line 9]
```

另外还有一类工具，能够分析程序在运行时的行为，从而找到软件中的竞争条件、死锁与 API 误用等现象，以及造成软件性能下降的原因。这些动态分析工具（参见第 59 条）可以找到一些很微妙的 bug，触发这种 bug 的指令，与 bug 本身的暴露地点通常隔得很远，以致大多数静态分析工具对此都无能为力。

与其他常见的动态分析工具一样，当你把软件放在这种工具下面运行的时候，其速度会慢好几个数量级，而且其消耗的内存，也会大幅增加。如果你想详细检视各条线程之间的依赖关系，那么必须付出这种代价。所幸现在的 64 位 CPU 很强大，能够使用数十 G 的内存来运作，甚至在笔记本电脑上面，也可以完成一些几年前无法想象的任务。如果你正在为很多遗留代码添加多线程功能，那么这些动态分析工具，或许可以帮助你找出数十种 bug，因此，即便运行的速度慢一点，占用的内存多一点，我想，也绝对是值得的。

下面举两个例子，用以演示怎样通过动态分析来寻找 OpenMP 及 POSIX Threads 代码中的错误。

首先看 OpenMP 的例子。下面是一段简单的程序，它会用多条线程来递增同一个 counter 变量。

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
int main()
{
      int i, counter;
#pragma omp parallel
      for (i = 0; i < 100000; i++)
            counter++;
      printf("counter=%d\n", counter);
      return 0;
}
```

如果你多次运行上述程序，那么就会看到多个不同的结果，这显然说明代码写得有问题。

```
$ ./race
counter=399757
$ ./race
counter=95561
$ ./race
counter=195790
```

我们在编译代码的时候开启调试选项，然后把编译好的程序，放在 Intel Inspector 工具下面运行，以寻找其中的死锁和数据竞争问题。该工具会生成图 8.1 这样的列表，其中列有两个数据竞争问题，由此我们显然可以看出：变量 i 没有在线程之间得到正确的共享，而且对 counter 变量的递增操作，也没有视为一项原子操作来执行。

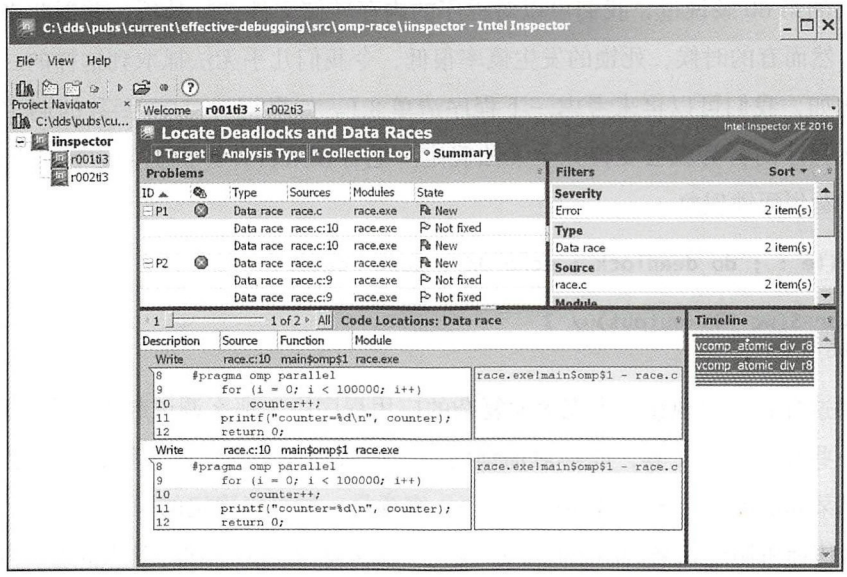


图 8.1　由 Intel Inspector 工具所发现的数据竞争问题

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

我们令每一条线程都拥有各自版本的变量 i，并且以原子操作的方式来对 counter 进行递增，这样就可以修复上述问题了。

```
#include <stdio.h>

static int i, counter;
#pragma omp threadprivate(i)

int main()
{
#pragma omp parallel
      for (i = 0; i < 100000; i++)
#pragma omp atomic
            counter++;
      printf("counter=%d\n", counter);
      return 0;
}
```

把程序修改好之后，我们可以看到：它所输出的值，总是 100000 的某个整数倍。

```
$ ./fixed
counter=800000
$ ./fixed
counter=800000
```

把修复好的程序放在 Intel Inspector 下面运行，是不会产生错误消息的。

本书的第 60 条说过，我们可以对程序的内存转储文件进行分析，并据此来调试死锁问题。然而有的时候，死锁的发生概率很低，令我们几乎无法抓取到这样的一份转储文件。比如，我们可以再来考虑一下程序清单 8.1。如果把对 sleep 函数的调用操作删掉，那么程序的行为就会变得完全无法确定。笔者在某台计算机上面运行了 6 1266 次之后，才看到死锁现象：

```
$ while : ; do deadlock ; echo OK ; done >output
^C
$ expr $(wc -l <output) / 3
61266
```

如果这种状况出现在一个庞大而复杂的应用程序中，那么调试起来就会特别困难。好在有一些动态分析工具，能够对加锁的顺序进行建模，并据此来探查这种潜在的问题。如果某算法能够分辨出这种加锁顺序，究竟是一种可以确定的顺序（例如，是一种通过消息传递机制建起来的固定顺序），还是一种无法确定的顺序（专业术语叫做偏序），那么，即便程序没有发生死锁，它也照样能够检测出潜在的死锁问题。

非卖品，仅供非商业用途或交流学习使用

Valgrind 工具包中的 Helgrind 工具，可以找出此类错误。它能够对以 POSIX Threads 原语所写的 C、C++ 及 Fortran 程序进行检查，进而找出其中的并发错误。我们只需把程序传给 valgrind 命令，并开启 --tool=helgrind 选项：

```
valgrind --tool=helgrind deadlock
```

该工具会输出一些信息，下面列出其中的一部分内容：

```
Helgrind, a thread error detector
Command: deadlock

Thread #3: lock order "0x600F40 before 0x600F80" violated

Observed (incorrect) order is: acquisition of lock at 0x600F80
   at 0x4C30616: pthread_mutex_lock (hg_intercepts.c:593)
   by 0x400834: alice (deadlock.c:24)
 followed by a later acquisition of lock at 0x600F40
   at 0x4C30616: pthread_mutex_lock (hg_intercepts.c:593)
   by 0x40085B: alice (deadlock.c:25)

Required order was established by acquisition of lock
   at 0x600F40
   at 0x4C30616: pthread_mutex_lock (hg_intercepts.c:593)
   by 0x40077B: bob (deadlock.c:14)
 followed by a later acquisition of lock at 0x600F80
   at 0x4C30616: pthread_mutex_lock (hg_intercepts.c:593)
   by 0x4007A2: bob (deadlock.c:15)

 Lock at 0x600F40 was first observed
   at 0x4C30616: pthread_mutex_lock (hg_intercepts.c:593)
   by 0x40077B: bob (deadlock.c:14)
 Address 0x600f40 is 0 bytes inside data symbol "m1"

 Lock at 0x600F80 was first observed
   at 0x4C30616: pthread_mutex_lock (hg_intercepts.c:593)
   by 0x4007A2: bob (deadlock.c:15)
 Address 0x600f80 is 0 bytes inside data symbol "m2"

ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 18 from 18)
```

上面这些信息的意思是说：Helgrind 工具首先发现，bob 线程已经确定了一种“先获取 m1 再获取 m2”的加锁顺序，但是稍后它又看到 alice 线程违背这个顺序，想要先获取 m2 再获取 m1。由于代码中没有适当的锁定或同步机制可以避免这种顺序错乱的现象，因此，Helgrind 就认为该程序的加锁顺序有问题，并把这个问题报告给我们。

非卖品，仅供非商业用途或交流学习使用

要点

- 用静态分析工具来梳理多线程代码，以寻找有可能发生的同步及锁定错误。
- 用动态分析工具来运行多线程代码，以寻找对 API 的误用现象，以及潜在的死锁与数据竞争问题。

第 63 条：把不确定的因素隔离出来，或将其移除

并发代码在执行的过程中缺乏确定性，这会给调试工作造成困难。除非你打算使用一些极其特殊的手段来调试（参见第 61 条），否则，对这样的代码进行调试，就相当于用沙子来盖楼一样：由于代码的行为总是不停地变化，因此你无法可靠地重现 bug、无法通过反复的尝试来详细观察 bug，也无法确定 bug 到底修复好了没有。除了本章已经讲过的方法及工具，还有两种办法能够用来应对这些不确定的因素，一种办法是把这些行为不确定的代码与其他代码隔开，还有一种办法是对这些代码进行适当的实现与配置，令其变得较为确定。

隔离法是把代码分成两个部分，从而帮助你解决复杂的并发软件中所包含的棘手问题。在这种名为 Humble Object 的模式下面，我们把带有不确定因素的并发代码，从程序的其他逻辑中隔离出来，也就是要把所有包含不确定因素的并发代码，都集中在一个很小的内核里面，同时把那些行为能够确定的代码，全都放在内核之外，从而令我们可以采用传统的工具与技术，对该内核进行直接而可靠的测试与调试。由于这个包含不确定因素的内核很小，所以能够轻松地接受正式或非正式的走查（walk-through）。我们既可以对该内核进行单独的排查，也可以将该内核与其他代码合起来排查。面对这个比较小的内核，我们应该能够看出其中有没有并发方面的问题，甚至能够给出一个并发问题不存在的论证（也就是证明内核中的代码是正确的）。由于并发代码的数量比较少，因此我们可以较好地把握住它的实质，当我们要把这些代码转换成伪代码，以便进行论证时，这一点显得尤为关键。此外，也可以考虑采用一些较为可靠的并发原语来表达这个内核的含义，并对其重写，以消除各种有可能出现并发错误的地方（参见第 66 条）。最后，把这两部分代码互相分离，还使得我们更有机会从架构上面深入地改善代码。

移除法是用行为可以预测的实体来替换那些行为不太确定的实体。比如，我们可以试着对多线程的组件进行配置，令其只用单线程来运行。（如果你使用的是自己可以控制

非卖品，仅供非商业用途或交流学习使用

的线程池，那么很容易就能做到这一点，反之，如果你使用的是第三方组件，那就比较困难了。）如果采用这种办法，那么在 Java 程序中，你可以把创建 ForkJoinPool 时所用的 parallelism 参数设为 1，而在使用了 SQLite 的 C/C++ 程序中，则可以通过 SQLITE_CONFIG_SINGLETHREAD 选项来对 SQLite 进行配置。用了这个办法之后，你在编写测试代码时，就知道程序应该产生什么样的结果了，而且也能知道程序会在什么样的情况下产生这样的结果，于是，你可以按照固定的步骤来反复进行调试，从而简化自己的测试与调试工作。在这种单线程的环境下，你可以把程序在按顺序执行的过程中所产生的那些问题调试好，尽管它没有办法帮你处理并发方面的 bug，但你至少能够知道某个 bug 是不是由并发问题所引发的。这些配置当然只应该在调试和测试的时候启用，而不应该带到生产环境中。

另外要说的是，如果某个对象是以异步方式来响应请求的，那么可以在程序内部针对该对象创建测试替身（test double）或模拟对象，令其等待响应的结果，从而使测试代码能够以同步的方式来获得这个结果。比如，如果我们在执行下面这个 shell 函数之前，率先设置了 TESTING 变量，那么该函数就会一直等着文件下载完毕，然后才继续往下执行，而不会像平常那样，采用异步的方式来获取这份文件。

```
fetch_file()
{
  local url="$1"
  local filename="$2"

  wget -q -O $filename $url &
  if [ "$TESTING" ] ; then
    wait
  fi
}
```

此外，如果程序会启动多条线程，并允许这些线程以异步的方式来执行操作，那么在测试这种程序时，你可以创建一份配置方案，使得该程序每次只执行一条线程，并且要等该线程执行完毕之后，再启动下一条线程。这样的行为，最好是在设计程序的时候就提前规划好。在这方面投入一些精力是值得的，因为这样做可以创建出更容易测试和调试的软件，从而减少程序将来发生错误的概率，并且使得我们在修复 bug 的时候，能够有更多的线索可供探查。

要点

- 把并发代码与其他代码隔开，使我们能够分别针对这两个方面来运用最为合适

非卖品，仅供非商业用途或交流学习使用

的调试工具与调试技术。

- 创建一份专供测试与调试所用的配置方案，并通过模拟对象等技术，把代码的行为固定下来，从而令程序在每次执行的时候，都能够展示出同样的效果。

第 64 条：检查资源争用情况，以解决与可伸缩性有关的问题

如果系统的整体性能（通常以延迟时间或吞吐量来衡量），没有能够随着当前可用的资源（如 CPU 的核心数量）而变化，那你就得看看其中有没有发生资源争用的问题。比如，系统中有没有那种可以改写为并发代码的函数、有没有涉及多个资源的加锁问题（详见下文），以及有没有内存缓存方面的问题（参见第 65 条）等。

现在来看程序清单 8.5 中的这段代码。该程序采用指定数量的线程来创建一个 map，使得该 map 包含特定数量的公钥 - 私钥对。

程序清单 8.5　采用多条线程来生成含有密钥对的 map

```
import java.security.*;
import java.util.concurrent.*;
import java.util.HashMap;

public class LockContention {
   static public void main(String[] args) {
      int nKeys = Integer.parseInt(args[0]);
      int nThreads = Integer.parseInt(args[1]);
      HashMap<PublicKey, PrivateKey> map =
         new HashMap<PublicKey,
         PrivateKey>();

      Runnable task = () -> {
         try {
            synchronized(map) {
               KeyPairGenerator keyGen = KeyPairGenerator
                  .getInstance("DSA", "SUN");
               SecureRandom random = SecureRandom
                  .getInstanceStrong();
               keyGen.initialize(2048, random);
               KeyPair pair = keyGen.generateKeyPair();
               map.put(pair.getPublic(), pair.getPrivate());
            }
         } catch (Exception e) {
           System.out.println("Generation failed: " + e);
         }
      };
```

非卖品，仅供非商业用途或交流学习使用


```
        ExecutorService executor = Executors
            .newFixedThreadPool(nThreads);
        for (int i = 0; i < nKeys; i++)
            executor.submit(task);
        try {
            executor.shutdown();
            executor.awaitTermination(5, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            System.err.println("Interrupted await: " + e);
        }
    }
}
```

在把密钥对的数量设为 1000 的前提下，用 4 条线程来运行该程序，所花的时间与用 1 条线程来运行的时候差不多。

```
$ time java LockContention 1000 4
real  0m11.106s

$ time java LockContention 1000 1
real  0m11.075s
```

在本例中，很容易就能看出系统性能无法随着线程数量而变化的原因，但如果你要调试的是一个更为庞大的系统，那么这样的问题就不太容易看出来了。对于此类问题来说，我们可以采用 profiling 工具来寻找引发资源争用的原因，也就是把多条线程因为争着获取同一份资源而陷入阻塞的情况找出来。在这些工具中，有两个工具颇为典型，一个是 Oracle 的 Java Flight Recorder，另一个是 Intel 的 VTune Amplifier。我们以前者为例来演示这种工具的用法。

首先，把程序放在 profiler 下面运行，以便将分析性能所需的数据收集起来：

```
$ java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder \
>   -XX:StartFlightRecording=name=test,dumponexit=true,\
>   filename=perf.jfr LockContention 1000 4
Started recording 1. No limit (duration/maxsize/maxage) in use.
```

你也可以把 profiler 连接到正在运行程序上面。对于本例所提到的 Java 程序来说，可以通过 Oracle 的 Java Mission Control 图形界面来进行连接。

下一步是对收集到的数据进行分析，一般来说，我们通过 profiler 的图形界面来做这种分析。在图 8.2 中，左上方的那幅小图，演示的是各条线程的执行时间，由于每条线程所获得的时间并不均等，因此这说明程序中可能有问题。再来看右上方的小图，该图演示了线程的阻塞情况，由此可知，线程受到阻塞的时间是很长的。单击阻塞情况最

非卖品，仅供非商业用途或交流学习使用

为严重的那条线程，并查看其栈跟踪信息，就可以发现：之所以会出现阻塞现象，是因为各条线程都在争着获取 HashMap 对象的锁定权。左下方和右下方的两张小图，再次印证了这一判断。从左下方的图中，可以看出程序发生延迟的原因，其中有多达 12.9 秒的延迟，都是线程受到阻塞所引发的；而右下方的图，则说明了加锁情况：程序在执行过程中所产生的那 12.9 秒延迟，全都花在了对 HashMap 的锁定上面。

要想解决这个与可伸缩性有关的问题，我们可以把 HashMap 改为 ConcurrentHashMap，以消除（过度的）同步行为，这样做可以使 4 线程的执行速度变为单线程的 3.2 倍。

```
$ time java NoContention 1000 4
real  0m3.503s
```

要点

- 用 profiling 工具来探查引发竞争现象的原因，以解决多线程代码中与可伸缩性有关的问题。

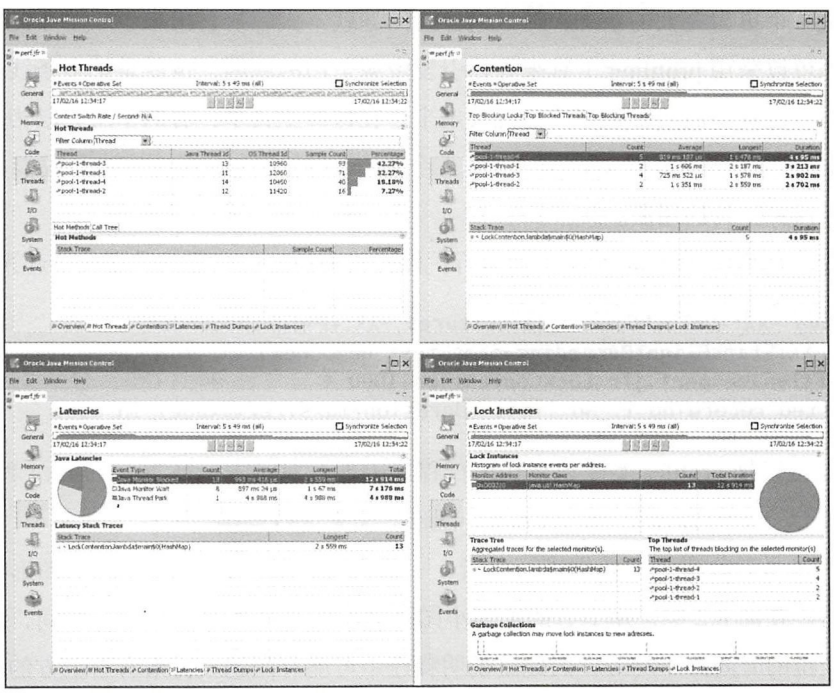


图 8.2　用 Java Flight Recorder 工具对竞争现象进行分析

第 65 条：用性能计数器寻找伪共享问题

下面是一段用 OpenMP 库所写的 C 语言代码，它会对 values 数组中的元素进行 8 次求和，每次求和时，都会将元素的值与 2 的整数次幂相除，后一次求和时所用的除数，是前一次的 2 倍。

$$\text{sum}_0 = \sum_{i=0}^{N} \left\lfloor \frac{\text{values}_i}{2^0} \right\rfloor, \ldots, \text{sum}_7 = \sum_{i=0}^{N} \left\lfloor \frac{\text{values}_i}{2^7} \right\rfloor$$

```
#include <omp.h>

#define N 100000000
#define NTHREADS 8
int values[N];
int
main(int argc, char *argv[])
{
   int tid;
   static int sum[NTHREADS];

#ifdef _OPENMP
   omp_set_num_threads(NTHREADS);
#pragma omp parallel private(tid)
   {
      tid = omp_get_thread_num();
#else
   for (tid = 0; tid < NTHREADS; tid++) {
#endif
      for (int i = 0; i < N; i++)
         sum[tid] += values[i] >> tid;
   }
}
```

如果启用 OpenMP 库的多线程功能，并以 8 条线程来运行该程序，那么它在 8 核的计算机上面，需要用 2603 毫秒才能执行完。

```
$ time ./sum-mp
real   0m2.603s
user   0m19.076s
sys    0m0.072s
```

如果在不启用 OpenMP 的情况下编译同一份代码，那么它将按照先后顺序完成 8 次求和，而这样的单线程程序，在同样一台计算机上，只需要 2249 毫秒就可以执行完。

```
$ time ./sum-seq
real  0m2.249s
user  0m2.208s
sys   0m0.040s
```

由此可见：用了多线程之后，程序的效率反倒下降了，这是个很奇怪的事情。用 8 条线程来执行相同的任务，应该比用 1 条线程快得多才对。

这种奇怪的现象是由伪共享（false sharing）问题造成的。由于范例代码并没有使用同步原语，因此该问题和同步原语无关。它是由另外一种同步机制所引发的，这种机制就是 CPU 核心的同步协议，也称为缓存一致性协议，该协议用来保证各线程总是能够看到相互一致的内存数据。多个 CPU 核心在共享内存的时候，每个核心都有可能把自己频繁使用的那部分内存内容，复制到自身的局部缓存中。如果某个核心要向局部缓存中的某个位置写入数据，而与该位置相对应的那个内存地址，同时也缓存到了另外一个核心的局部缓存中，并且那个核心又想要读取该地址中的数据，那么 CPU 就会触发一套复杂的机制，以确保这两个核心都能在其各自的缓存中看到相同的内容——这就是同步协议所要解决的问题。而刚才那个范例程序之所以运行得较为缓慢，正是由于缓存同步协议想要使所有的线程都能够看到相同的 sum 数组。一般情况下，每条线程都应该各自操作不同的内存区域，而不应该去干涉其他线程所操作的那块区域，然而在本例中，由于 sum 数组很小，因此每一个 CPU 核心都有可能把该数组放到相同的缓存区块中，于是就造成了 sum 数组在多个核心之间共享的局面。即便每条线程都只操作该数组中的一个元素，它们也依然共享着整个 sum 数组，这就是伪共享一词的含义所在。每个 CPU 核心在对 sum 数组中的某个元素进行操作时，都要把缓存中的整个数组推送到（速度相对较慢）主内存中，使得其他 CPU 核心也能够看到同样一份数组，而在读取该数组中某个元素的值时，也必须从主内存里面获取。（缓存同步机制有多种实现方式，但是这些方式都会产生明显的开销。）

要想有效地调试这种伪共享问题，我们可以探查 CPU 的性能计数器。这些计数器会把与 CPU 性能有关的事件记录下来，例如，执行的指令数量以及缓存未命中的次数。这样的工具包括 Visual Studio 的 Concurrency Visualizer 扩展、Intel 的 VTune Performance Analyzer 以及 Linux 的 perf 命令等，它们都可以用来寻找程序在运行时所遭遇的伪共享（以及其他一些）问题，并帮助你确定其根源。

下面演示怎样通过 perf 命令来统计刚才那个程序的末级缓存（last-level cache，简

称 LLC）未命中次数（该次数在 perf 命令中，以 LLC-loads 表示）。如果程序对缓存一致性协议的触发次数比较多，那么 LLC-loads 的读数就会随之增大（对于本例来说，由于发生了伪共享问题，因此这个数字显得特别大）。

```
$ perf stat --event=LLC-loads ./sum-seq
 Performance counter stats for './sum-seq':
          17,830    LLC-loads
     2.223350547 seconds time elapsed
$ perf stat -e LLC-loads ./sum-mp
 Performance counter stats for './sum-mp':
      49,264,883    LLC-loads
     2.547188760 seconds time elapsed
```

接下来，我们要用 perf 命令记录相关的事件，以便把受到伪共享问题所影响的那些代码找出来。

```
perf record --event=LLC-loads ./sum-mp
```

你也可以把 perf 连接到已经运行的程序上面。

最后，我们用 perf annotate 命令来查看刚才记录到的结果，以寻找与缓存未命中现象关系最大的代码。perf 工具提供了一套图形界面，适合用来分析大型的程序，然而对于本例来说，文本形式的输出信息，就足以说明问题了。我们可以看到，在所有的末级缓存加载（last-level cache loads）事件中，有百分之 54.90（=25.03+14.23+14.83）的事件，都是由那行对 sum 数组进行写入的代码所引发的。

```
 Percent |  Source code & Disassembly of sum-mp for LLC-loads
-----------------------------------------
       :    Disassembly of section .text:
       :       {
       :           tid = omp_get_thread_num();
  0.00 :     4006eb:    callq 400560 <omp_get_thread_num@plt>
  0.00 :     4006f0:    mov   %eax,-0x8(%rbp)
       :    #else
       :       for (tid = 0; tid < NTHREADS; tid++) {
       :    #endif
       :           for (int i = 0; i < N; i++)
  0.00 :     4006f3:    movl  $0x0,-0x4(%rbp)
  0.19 :     4006fa:    cmpl  $0x5f5e0ff,-0x4(%rbp)
  2.42 :     400701:    jg    400738 <main._omp_fn.0+0x59>
       :               sum[tid] += values[i] >> tid;
  0.36 :     400703:    mov   -0x8(%rbp),%eax
  0.79 :     400706:    cltq
  0.22 :     400708:    mov   0x600bc0(,%rax,4),%edx
```

非卖品，仅供非商业用途或交流学习使用

```
  25.03 :    40070f:    mov   -0x4(%rbp),%eax
   0.55 :    400712:    cltq
   0.37 :    400714:    mov   0x600c00(,%rax,4),%esi
   2.21 :    40071b:    mov   -0x8(%rbp),%eax
   2.00 :    40071e:    mov   %eax,%ecx
   0.06 :    400720:    sar   %cl,%esi
   2.68 :    400722:    mov   %esi,%eax
   0.06 :    400724:    add   %eax,%edx
   0.83 :    400726:    mov   -0x8(%rbp),%eax
  14.23 :    400729:    cltq
  14.83 :    40072b:    mov   %edx,0x600bc0(,%rax,4)
```

根据上述结论，我们可以对程序做出修改，用基于栈的变量来保存在数组求和过程中所得到的累加值，等到求和完毕之后，再把该值写入 sum 数组。

```
#include <omp.h>

#define N 100000000
#define NTHREADS 8
int values[N];
int
main(int argc, char *argv[])
{
   int tid;
   static int sum[NTHREADS];

   omp_set_num_threads(NTHREADS);
#pragma omp parallel private(tid)
   {
      int local_sum = 0;
      tid = omp_get_thread_num();
      for (int i = 0; i < N; i++)
         local_sum += values[i] >> tid;
      sum[tid] = local_sum;
   }
}
```

修改之后的并行程序，其运行速度是单线程版本的 4 倍。

```
$ time sum-mp-noshare
real  0m0.553s
user  0m4.276s
sys   0m0.072s
```

要点

用 profiling 工具来监控性能计数器，以寻找并解决程序中的伪共享问题。

非卖品，仅供非商业用途或交流学习使用

第 66 条：考虑用更为高级的抽象机制来重写代码

通过本章其他几条所讲的内容，大家可以看到，底层的多线程代码是很难操控的。除了竞争条件、死锁、活锁（livelock，如果代码处于这种状态，那么它只会空转，无法完成任何工作）以及性能不佳等常见的问题，还会涌现许多的 bug。如果你所遭遇的 bug 看上去无法解决，那么有时不妨考虑把现有的代码丢掉（或者说，至少把其中的并发组件丢掉），改用更为高级的解决方案来做。（还有一种办法，是采用另外一种编程语言来实现，参见第 47 条。）从架构的角度来看，有很多种高级的方案，都可以令程序稳定而高效地运行在多个 CPU 核心上面，其中包括使用消息总线或工作队列等。然而这些方案并不在本书的讨论范围之内，我们这里想要讨论的，是一些代码技巧，这些技巧或许可以帮你绕开并行方面的一些 bug，它们会请其他软件来处理与多核 CPU 有关的事宜，同时又令自身可以享受到由多核所带来的好处。与其他一些奇妙的技巧一样，这些技巧也不能保证绝对有效，然而一旦见效，其结果就会非常显著。有时我们只需要花费少量的精力（此外可能还得花一些钱），就可以获得很好的提速效果。

从最为宏观的层面来看，如果你能找到现成的中间件来完成并发方面的任务，那么很容易就可以将其分配到多个 CPU 核心上面。对于那种需要处理 Web 请求或 SQL 语句的应用程序来说，情况正是如此。Web 应用程序服务器会把工作量分割到多条线程或进程中，进而充分地利用 CPU 的各个核心。你只需把自己的应用程序，放在 JavaEE 或 Node.js 这样的服务器框架里面来运行就可以了。对于需要处理 SQL 语句的应用程序来说，你可以把这些语句转交给一套成熟的关系型数据库管理系统，那些系统的优化引擎，会对 SQL 查询进行优化，并设法将查询工作分配到多个 CPU 核心上面，比如，它可能会把每一条对表格进行过滤的 WHERE 子句，都单独放到某个核心上面去执行。此时你的主要职责，只不过是把任务尽量交给数据库系统来完成。

还有一种宏观的方式，也可以将工作分配到多个 CPU 核心上面，那就是把工作交给操作系统，由操作系统来将其分割到多个独立的进程中。有些应用程序可能是以管道加过滤器的架构来编写的，其中一条进程产生输出信息，另一条进程消耗这些信息（参见第 22 条）。这种管道式的写法，是随着 Unix shell 而得以普及的，你可以用这种写法轻松地拼接一系列进程，而无需担心这些进程究竟怎样与多个 CPU 内核对应起来，因为目前凡是比较好的操作系统，都会自动实现这一功能。比如，如果软件需要把数据文件的压缩格式由 bzip2 转换成 gzip，那么运行下面这条命令管道就可以了：

非卖品，仅供非商业用途或交流学习使用

```
bzip2 -dc data.bz2 | gzip -c >data.gz
```

这条管道会以并发的方式来执行 bzip2 解压程序及 gzip 压缩程序。通过下面的测评结果，我们可以看到，对于一份 70MB 的压缩文件来说，按照先后顺序执行这两个命令，要花费 41 秒，但如果通过管道来执行，那么只需 27 秒就可以完成。

```
$ time { bzip2 -d data.bz2 ; gzip data ; }
real  0m41.367s
user  0m40.325s
sys   0m0.919s
$ time bzip2 -dc <data.bz2 | gzip -c >data.gz
real  0m27.444s
user  0m40.886s
sys   0m1.278s
```

如果你要调试的程序，是按照从先到后的顺序来处理多个数据块的（例如，多个文件、多行文本或多条记录），那么可以通过 GNU parallel 工具，将这些处理任务轻松地划分到各个 CPU 核心上面。你只需要把待处理的数据块发给这个工具就可以了，它会自动运行很多任务，以求充分利用这些 CPU 核心。比如，如果你的软件需要根据照片生成缩略图，那么可以像下面这样，通过 parallel 命令来调用 JPEG 解压缩及压缩程序：

```
ls *.jpg | parallel 'djpeg -scale 1/16 {} | cjpeg >thumb/{}'
```

通过下面这些评测数据，我们可以看出，这种做法能够令程序的运行时间减半。

```
$ time ls *.jpeg |
> xargs -I '{}' sh -c 'djpeg -scale 1/16 {} | cjpeg >thumb/{}'
real  0m10.493s
user  0m6.428s
sys   0m4.360s

$ time ls *.[Jj]* |
> parallel 'djpeg -scale 1/16 {} | cjpeg >thumb/{}'
real  0m4.149s
user  0m8.384s
sys   0m6.696s
```

如果你要调试的代码，其职责是接受一份庞大的输入文件，并将这项大任务划分成多项小任务，那么 parallel 工具可以帮你进行这种划分。（笔者曾经试着用低级的线程操作或异步 I/O 实现过这样的功能，实现起来很麻烦，而且很难做好。）

并不是所有的处理任务都可以像刚才那样，建模成一系列彼此无关的步骤，并通过 parallel 命令，用管道或独立进程的方式来加以运行，因为在很多情况下，任务的各个

非卖品，仅供非商业用途或交流学习使用

步骤之间，其实是有依赖关系的。面对这样的任务，我们可以（粗略地）把其中某些步骤的先决条件，以及文件之间的依赖关系，描述到 Makefile 中，然后交给 Unix 的 make 工具来处理。版本较新的 make 工具可以指定 -j 参数，以便在满足依赖关系的前提下，尽可能多地把大任务拆分成多个小任务（job），并加以平行地执行。这种做法，通常是用来在编译系统的时候缩短编译时间的，然而你也可以用它来做别的事情，例如，对电影进行渲染，或是对庞大的数据进行处理等。如果在一台 8 核的计算机上面编译 4.5 版的 Linux 内核，那么启用了并发功能的 make 命令（通过 -j 8 参数来开启），可以把编译所耗的时间，从 14 分钟缩短为 3 分钟。

```
$ time make >/dev/null
real  14m18.053s
user  11m35.152s
sys   1m41.608s

$ time make -j 8 >/dev/null
real  3m12.827s
user  20m32.272s
sys   2m35.600s
```

parallel 命令会把指定的流程分别运用到各个数据块上面，以实现并行，你也可以通过 map-reduce（映射 – 归纳、映射 – 化简）及 filter-reduce（过滤 – 归纳、过滤 – 化简）技术来把这个思路运用在自己的应用程序里面。这些技术可以通过某个函数来对 vector 等容器中的元素进行映射（也就是修改每一个元素）或过滤（也就是选出其中的某些元素），然后通过另外一个函数，把映射或过滤的结果，归纳成单独的一项内容。如果你打算运用的函数是较为耗时的，那么这种操作就有助于提升程序的效率，因为它可以把应用程序级别的线程，分割到多个 CPU 核心中去执行。比如，可以用 java.util 包的 Collection.parallelStream 方法或 QtConcurrent C++ 库的相关函数来执行这样的操作。你只需要使用适当的容器，并指定与对应 API 相兼容的映射函数、过滤函数或归纳函数即可。如果要并行处理的任务比较大，而且是粗粒度的任务，那么可以将其完全交给 Apache Hadoop 等分布式的处理框架来做。

有很多较为常见且工作量较为庞大的任务，都可以通过相应的程序库来实现并行，这些程序库经过了手动的调节，可以充分发挥出多核 CPU 的能力。AMD 及 Intel 等处理器厂商，都提供有这样的程序库，可以用来对音频与视频进行编码和解码，可以用来处理图像、语音以及通用的信号，还可以用来进行加密、压缩及渲染。此外，也很多更为专门的程序库可供选用。比如，实现了 BLAS（Basic Linear Algebra Subprograms，基

非卖品，仅供非商业用途或交流学习使用

础线性代数程序集）标准的 ATLAS（Automatically Tuned Linear Algebra Software）库，就属于此类。它会在构建的时候，根据硬件所具备的能力来动态地调节程序库中的组件，令二者相互匹配。另外一个例子是 NAG Numerical Components Library for SMP and multicore，它实现了平行化的数值计算与统计算法。如果你能把问题用矩阵操作的形式表达出来（其实有很多问题都可以这样表达），那么或许就可以把原来那些有 bug 的代码删掉，并通过调用该程序库来解决问题。

适当地采用这种程序库来进行平行运算，可以提升程序的性能，而且能够避开编写并发程序时所遇到的那些难题。下面举一个例子，以演示这种程序库所起到的效果。这是一个 R 语言的程序，用来计算 10000 × 10000 矩阵的逆矩阵。

```
#!/usr/bin/env Rscript

# Matrix size
n <- 10000

# Create a square matrix of random numbers
m <- replicate(n, rnorm(n))

# Calculate the matrix inverse
r <- solve(m)
```

如果用针对单核计算机所构建的某个 ATLAS 程序库来运行该程序，那么大概需要 3 分钟的时间。

```
$ time ./solve.R
real  3m10.285s
user  3m8.476s
sys   0m1.800s
```

如果改用另外一个针对多核计算机所优化的 BLAS 库来运行这个 R 程序，那么只需要 1 分钟左右就可以执行完毕。

```
$ time ./solve.R
real  1m1.995s
user  3m11.876s
sys   0m3.256s
```

另外一种办法，是使用那种容易发挥多核优势的编程语言或编程风格。在这一点上，函数式编程的好处较为明显，因为在以这种风格所写的程序中，程序块之间是不会互相干扰的。因此，如果你开发的是 Java 程序，那就从 lambda 表达式和 stream 入手；如果你开发的是那种必须与现有 API 相结合的程序，那就尝试一种与对应的框架有

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

关的函数式编程语言，例如，适用于 JVM、.NET 和 JavaScript 的 Clojure 语言，适用于 JVM 的 Scala 语言，以及适用于 .NET 的 F# 语言等。此外，还有一类更为彻底的做法，例如，使用像 Haskell 这样的纯函数式语言、使用像 R 这样专门给特定领域所设计的语言，或是使用像 Erlang 这样从设计上明确支持并发的语言。如果你要从头开始设计一套数据处理量很大的系统，而且这套系统不太需要进行 API 方面的交互，那么此类做法就是可行的。如果你要从头开始为一款有 bug 的软件重新做架构，那么除了上述办法，还有一种思路也可以发挥多核的优势，那就是采用 Vert.x 这样的反应式事件驱动框架。

现在我们看看怎样通过平行地调用函数来提升程序的性能。下面这段代码会对长度为 5 的每一个英文单词进行检查，看看该单词中的字符能否以另外一种排列方式构成其他的单词，然后将符合此条件的全部单词放在一份列表中（比如，“trust”一词就符合该条件，因为我们可以对该单词中的 5 个字母进行排列，从而构成另外一个英文单词“strut”）。程序清单 8.6 中的 R 程序可以找到这些单词。

程序清单 8.6　在 R 程序中平行地运用函数

```
#!/usr/bin/env Rscript
library(combinat)
library(parallel)

# Read in a file of English words
words <- readLines('/usr/share/dict/words')

# Obtain five letter words
flw <- words[nchar(words) == 5]

# Return words consisting of permutations of the passed word
word.permutations <- function(w) {
  # Obtain all character permutations
  p <- lapply(strsplit(w, NULL), permn)
  # Convert permutations to list of words
  r <- sapply(unlist(p, recursive=FALSE), paste, collapse="")
  # Remove permutations resulting in the original word
  new <- r[r != w]
  # Return the intersection of the two sets
  intersect(flw, new)
}

# Generate list of words that are permutations of others
p <- unlist(lapply(flw, word.permutations))
```

这个 R 程序的全部工作，其实都是由 word.permutations 函数来完成的。在由 5 个字符所组成的某单词上面调用该函数，即可列出由这 5 个字符的其他排列方式所构成的

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

单词。

```
> word.permutations('teams')
[1] "mates" "meats" "steam" "tames"
```

有了这个函数之后，R 程序就可以通过 lapply 函数，把它运用到长度为 5 的所有单词（也就是由 flw 所表示的那份列表）上面，从而把其中满足条件的那些单词找出来。如果把 lapply 改为平行化的 mclapply 版本，那么只需花费不到四分之一的时间，就可以计算出结果。

```
> system.time(lapply(flw, word.permutations))
   user  system elapsed
 20.896   0.000  20.896

> system.time(mclapply(flw, word.permutations, mc.cores=8))
   user  system elapsed
 42.976   0.268   4.623
```

最后要说的是，有很多编程框架都在通过引入更为高级的原语来解决与低级并发代码有关的 bug。因此，在条件允许的情况下，可以试着改用这些原语来编写代码，而不用再与那些低级的多线程代码相纠缠，因为要想修复那些混乱的代码，是很困难的。下面以 Java 的并发工具包为例来概述这种用法：

- 用 Executor 框架的 Executor 与 ExecutorService 接口（参见第 64 条），取代那些对低级线程进行管理的代码。
- 用 CountDownLatch、CyclicBarrier、Exchanger、Phaser 及 Semaphore 类，取代那些以原始的 synchronized 代码块所实现的类似功能。
- 用 java.util.concurrent 包所提供的并发集合以及具有弱一致性的迭代器来取代自编的同步机制、java.util 中的某些内容，以及 Collections 中的同步适配器（synchronization adapter）方法。可供考虑的类有：ArrayBlockingQueue、BlockingDeque、BlockingQueue、ConcurrentHashMap、ConcurrentLinkedDeque、ConcurrentLinkedQueue、ConcurrentMap、ConcurrentNavigableMap、ConcurrentSkipListMap、ConcurrentSkipListSet、CopyOnWriteArrayList、CopyOnWriteArraySet、DelayQueue、LinkedBlockingDeque、LinkedBlockingQueue、LinkedTransferQueue、PriorityBlockingQueue、SynchronousQueue 与 TransferQueue。
- 如果任务之间的关系较为复杂，那么可以考虑用 CompletableFuture 类协调这些任务的并行情况。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 采用并行 Stream、FutureTask 及 lambda 表达式来描述那些将要并发执行的过滤与映射操作。

我们来看程序清单 8.7 中的这个程序，它会根据第一个参数的值，对某份文件进行处理，把其中的各项 IP 地址都解析为对应的主机名称，并将其输出。比如，如果文件里面包含 8.8.8.8 这个 IP 地址，那么程序就会输出 google-public-dns-a.google.com。

程序清单 8.7　基于 Stream 的 IP 地址解析程序

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.CompletableFuture;
import java.util.stream.Collectors;
import java.util.List;

/** Resolve IP addressed in file args[0] using 100 threads */
public class Resolve100CF {
   /** Resolve the passed internet address into a name */
   static String addressName(String ipAddress) {
      try {
         return InetAddress.getByName(ipAddress).getHostName();
      } catch (UnknownHostException e) {
         return ipAddress;
      }
   }

   public static void main(String[] args) {
      Path path = Paths.get(args[0]);
      // Create pool of 100 threads to compute results
      ForkJoinPool fjp = new ForkJoinPool(100);

      try {
         // Obtain list of lines
         List<CompletableFuture<String>> list =
            Files.lines(path)
            // Map lines into a future task
            .map(line -> CompletableFuture.supplyAsync(
                     () -> addressName(line), fjp))
            // Collect future tasks into a list
            .collect(Collectors.toList());
         // Wait for tasks to complete, and print the result
         list.stream().map(CompletableFuture::join)
            .forEach(System.out::println);
```

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            } catch (IOException e) {
                System.err.println("Failed: " + e);
            }
        }
    }
```

由于这段程序利用了 Stream 及 CompletableFuture 类，因此它不需要对线程及其运行结果进行手工的管理。对于含有 1000 个 IP 地址的文件来说，该程序只需 37 秒就可以处理完，而那种按顺序执行的实现方式，则要用 48 分钟才能完成，前者的速度是后者的 78 倍。

```
$ time java ResolveFuture 1000.ip >1000p.name
real  0m37.465s
user  0m0.015s
sys   0m0.000s

$ time java ResolveSequential 1000.ip >1000s.name
real  48m40.036s
user  0m0.000s
sys   0m0.015s
```

按顺序执行的方式之所以很慢，其原因在于：DNS 查询会引发很长的 I/O 延迟。因此，对这样一种任务加以平行化的处理是非常合适的，而且范例程序还避开了低级的并发原语。

要点

- 为了避开并发方面的问题，可以考虑用特殊的编程语言、处理流程、工具、框架或程序库等更为高级的办法来重新实现那些含有 bug 的并发代码。

非卖品，仅供非商业用途或交流学习使用

网上资源

Ansible	http://www.ansible.com/
Ant	http://ant.apache.org/
Apache Hadoop	http://hadoop.apache.org/
Apache HTTP Server	http://httpd.apache.org/
Apache JMeter	http://jmeter.apache.org/
Apache log4j	http://logging.apache.org/log4j
"The Art of Command Line"	https://github.com/jlevy/the-art-of-command-line
AutoHotkey	http://www.autohotkey.com/
AutoKey	https://en.wikipedia.org/wiki/AutoKey
Automator	https://en.wikipedia.org/wiki/Automator(software)
Black Duck Open Hub Code Search	https://code.openhub.net/
Boost.Log v2	http://www.boost.org/doc/libs/1_59_0/libs/log/doc/html/index.html
Bugzilla	https://www.bugzilla.org/
Bunyan	https://github.com/trentm/node-bunyan
bytecode	http://andrei.gmxhome.de/bytecode/
Byteman	http://byteman.jboss.org/
CFEngine	http://www.cfengine.com/
cgdb	http://cgdb.github.io
Chef	http://www.chef.io/
Chronon	http://chrononsystems.com/
collectd	https://collectd.org/
CppUnit	http://cppunit.sourceforge.net
Crittercism	http://www.crittercism.com/
ctags	https://en.wikipedia.org/wiki/Ctags
cURL	http://curl.haxx.se/
Cygwin	https://cygwin.com/
DDD	http://www.gnu.org/software/ddd
dmalloc	http://dmalloc.com/
Docker	https://www.docker.com/
Dynamic program analysis	https://en.wikipedia.org/wiki/Dynamic_program_analysis
"Eradicating Non-Determinism"	http://martinfowler.com/articles/nonDeterminism.html
FindBugs	http://findbugs.sourceforge.net/
GCC	https://gcc.gnu.org/
gdbinit	https://gist.github.com/CocoaBeans/1879270

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Gerrit	https://www.gerritcodereview.com/
ghi	https://github.com/stephencelis/ghi/
Git	https://git-scm.com/
GitHub	https://github.com/
GitLab	https://about.gitlab.com/
GNU Classpath	http://www.classpath.org/
GNU parallel	http://www.gnu.org/software/parallel/
Graphviz	http://graphviz.org/
Homebrew	http://brew.sh/
How To Ask Questions The Smart Way	http://www.catb.org/~esr/faqs/smart-questions.html
Humble Object	http://xunitpatterns.com/Humble%20Object.html
Intel Inspector	https://software.intel.com/en-us/intel-inspector-xe
Jalangi	https://www.eecs.berkeley.edu/~gongliang13/jalangi_ff/index.html
JProfile	http://www.ej-technologies.com/products/jprofiler/overview.html
jq	http://stedolan.github.io/jq/
JRuby	http://jruby.org/
JSFiddle	https://jsfiddle.net/
Launchpad	https://launchpad.net/
libmicrohttpd	http://www.gnu.org/software/libmicrohttpd/
LLVM	http://llvm.org/
Log4j	http://logging.apache.org/
LTTng	http://lttng.org/
Lua	http://www.lua.org/
Magic Number	https://en.wikipedia.org/wiki/Magic_number_%28programming%29
make	https://en.wikipedia.org/wiki/Make(software)
Maven	https://maven.apache.org/
Mono	http://www.mono-project.com/
mruby	http://www.mruby.org/
Nagios	https://www.nagios.org/
New Relic	http://newrelic.com/
OpenJDK	http://openjdk.java.net/
OTRS	http://www.otrs.org/
Outwit	http://www.spinellis.gr/sw/outwit/
PLCrashReporter	https://www.plcrashreporter.org/
Process Monitor	https://technet.microsoft.com/en-us/sysinternals/processmonitor
Program Record/ Replay Toolkit	http://www.pintool.org/
Pull request	https://help.github.com/articles/using-pull-requests/
Puppet	http://www.puppetlabs.com/
Python Tutor	http://www.pythontutor.com/
qmcalc	https://github.com/dspinellis/cqmetrics
Raspberry Pi	http://raspberrypi.org/
Redmine	http://www.redmine.org/
Reproducible builds	https://reproducible-builds.org/

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

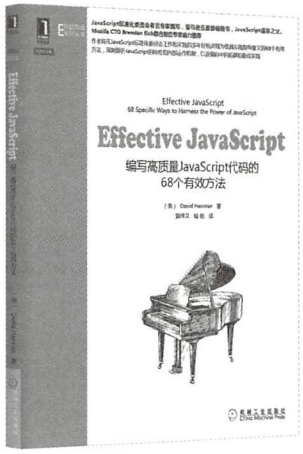
RRDtool	http://oss.oetiker.ch/rrdtool/
Rubinius	http://rubinius.com/
Rule of three	https://en.wikipedia.org/wiki/Rule_of_three_%28C%2B%2B_programming%29
saleae	https://www.saleae.com
Salt	http://saltstack.com/
Selenium	http://www.seleniumhq.org/
SourceLair	https://www.sourcelair.com/
Splunk MINT	http://www.splunk.com
spy-js	http://spy-js.com/
sscce.org	http://sscce.org/
Stack Overflow	http://www.stackoverflow.com
StackExchange	http://www.stackexchange.com
Static analysis tools	https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
tcpdump	http://www.tcpdump.org/
TeamViewer	https://www.teamviewer.com/
"TestDouble"	http://martinfowler.com/bliki/TestDouble.html
Trac	http://trac.edgewall.org/
Unit testing frameworks	https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
Valgrind	http://valgrind.org/
Vert.x	http://vertx.io/
vim	http://www.vim.org/
VisualVM	https://visualvm.java.net/
VLC	http://www.videolan.org/index.html
Wat	https://www.destroyallsoftware.com/talks/wat
Winston	https://github.com/flatiron/winston
Wireshark	https://www.wireshark.org/
zzuf	http://caca.zoy.org/wiki/zzuf

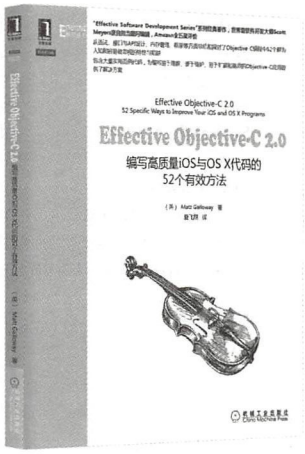
请访问英文版原书网站 www.spinellis.gr/debugging 以查看最新的列表。

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

推荐阅读





Effective Java中文版 第2版

作者：Joshua Bloch ISBN：978-7-111-25583-3 定价：52.00元

本书介绍了在Java编程中78条极具实用价值的经验规则，这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。通过对Java平台设计专家所使用的技术的全面描述，揭示了应该做什么，不应该做什么才能产生清晰、健壮和高效的代码。

本书中的每条规则都以简短、独立的小文章形式出现，并通过例子代码加以进一步说明。本书内容全面，结构清晰，讲解详细。可作为技术人员的参考用书。

Effective JavaScript: 编写高质量JavaScript代码的68个有效方法

作者：David Herman ISBN：978-7-111-44623-1 定价：49.00元

作者将在JavaScript标准化委员会工作和实践的多年经验浓缩为极具实践指导意义的68个有效方法，深刻辨析JavaScript的特性和内部运作机制，以及编码中的陷阱和最佳实践。

Effective Objective-C 2.0：编写高质量iOS与OS X代码的52个有效方法

作者：Matt Galloway ISBN：978-7-111-45129-7 定价：69.00元

本书是世界级C++开发大师Scott Meyers亲自担当顾问编辑的“Effective Software Development Series”系列丛书中的新作，Amazon全五星评价。从语法、接口与API设计、内存管理、框架等7大方面总结和探讨了Objective-C编程 中52个鲜为人知和容易被忽视的特性与陷阱。书中包含大量实用范例代码，为编写易于理解、便于维护、易于扩展和高效的Objective-C应用提供了解 决方案。

作者简介

Diomidis Spinellis

雅典经贸大学管理科学与技术系教授，经常讲授复杂系统的开发与调试技术。他是Google的高级软件工程师，也是FreeBSD committer团队的成员，并贡献了一些随OS X及BSD Unix系统而发布的代码。此外，他还开发了UMLGraph及CScout等流行的开源软件。Spinellis所写的《Code Reading》和《Code Quality》曾获得软件开发生产力奖。他是ACM及IEEE的资深成员。

封面设计：王建敏

Effective Debugging: 66 Specific Ways to Debug Software and Systems

每位软件开发者及IT人士都明白高效率的调试是多么重要。在开发者的日常工作中，最耗时间的环节通常就是程序的调试，而且相关的调试技术与技巧可能要花很长时间才能掌握。在本书中，Diomidis Spinellis对最为有用的调试方法、策略、技巧及工具进行系统的归类、整理与演示，以帮助有经验的程序员迅速掌握这些知识。

通过阅读本书，你将学到：

- 适用于各种软件故障的宏观策略及方法
- 在编写、编译及运行代码时可以使用的具体技术
- 如何最为充分地利用调试器所具备的能力
- 值得花时间来学习的一些通用技能与工具
- 一些高端的思路与技术，可以帮你摆脱调试时所遇到的困境以及过于复杂的bug
- 怎样把程序编写得更易于调试
- 在调试多线程的、异步的以及嵌入式的代码时所使用的特殊办法
- 怎样通过改进软件的设计、构建及管理来避开某些bug

Pearson
www.pearson.com

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/软件工程
ISBN 978-7-111-56889-6
9 787111 568896
定价：59.00元